AFRL-RI-RS-TR-2015-158

RESILIENT SOFTWARE SYSTEMS

VANDERBILT UNIVERSITY

JUNE 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JUNE 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* DEC 2012 – DEC 2014

4. TITLE AND SUBTITLE RESILIENT SOFTWARE SYSTEMS	**5a. CONTRACT NUMBER** FA8750-13-2-0050
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 63781D
6. AUTHOR(S) Gabor Karsai, Abhishek Dubey, Nag Mahadevan	**5d. PROJECT NUMBER** ASET
	5e. TASK NUMBER 12
	5f. WORK UNIT NUMBER VU
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Vanderbilt University, 110 21ST Avenue S Ste 937, Nashville TN 37203-2416	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-158

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2015-3138
Date Cleared:22 JUN 2015

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project developed an approach to modeling resilient software systems and a concrete approach to resilience in component-based software systems. Two techniques were developed for modeling resilient software architectures: one based on conventional patterns, and another one based on a more flexible and general approach. While both of them were useful, the first one suffered from usability problems, while the second one was more generic and less complex. Vanderbilt analyzed various resilience scenarios using a template developed. The template allows the documentation of scenarios and can assists a system architect in developing solutions. They developed an approach to facilitate run-time resilience through a resilience engine. The method encodes the configuration space of the system in a mathematical model and then uses a general purpose constraint solver to compute solutions that are alternative configurations of the system that can have failing components. The approach has been prototyped in a demonstration package. It is the conclusion that model-based development and engineering is necessary for such systems, due to the inherent potential complexity of these systems.

15. SUBJECT TERMS

Metamodel, Resilience Engine, Software Generators, Resilient Data Model

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** WILLIAM McKEEVER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	71	**19b. TELEPHONE NUMBER** *(Include area code)* N/A

TABLE OF CONTENTS

LIST OF FIGURES

1.0 SUMMARY

In today's complex technical systems software plays an essential role both as the provider of functionality and as a universal system integrator. Hence the survivability and resilience of the system (or the system of systems) critically depends on software: if any component, be it hardware or software, of a system fails we expect that the system recovers and survives with the help of algorithms embedded in software. Furthermore, the complexity of the systems has progressed to the point that zero-defect systems (containing both hardware and software) are very difficult and expensive to build – thus the system has to be prepared to handle latent defects that make it into deployment.

The goal of the project was to address the engineering challenge: How to build software intensive systems that anticipate change: uncertain environments, faults, updates, and exhibit resilience: they survive and adapt to changes, while being dependably functional?

The project has addressed this issue by developing the following artifacts:

1. A domain-specific modeling language for modeling resilient software architectures. The language defined by a metamodel, and is supported by a modeling tool based on the Generic Modeling Environment.
2. A collection of resilience design patterns for software architecture. The design patterns have been incorporated into the modeling language.
3. A collection of resilience scenarios that capture typical cases and techniques for resilient software systems.
4. A software generator tool for translating architecture models into deployment plans and a constraint-based representation of the resilient architecture. The deployment plans can be used by a run-time engine to configure and instantiate a running software system, while the representation can be used to facilitate reconfiguration of the running system.
5. A metric for determining the resilience of a software architecture. Additionally, a tool has been developed that calculates the resilience metric from a model of the architecture. The calculations have been used to evaluate and compare architectural alternatives.
6. A method for calculating the performance of a distributed architecture, given a network topology, the performance of the network link as a function of time, the communication requirements of software components, a model of the software architecture, and a specific deployment (mapping) of the software architecture on the network. The method has been implemented in a tool, and has been tested using small examples from the domain of distributed communications space architectures.
7. A run-time reconfiguration engine based on an Satisfiability Modulo Theory (SMT) constraint-solver and a fault-tolerant, high-performance database that is capable of the run-time dynamic reconfiguration of a component-based software architecture upon the detection of faults. The engine performs on-line verification on the component architecture, as it verifies that all system goals are satisfied, and it has been tested with respect to performance impact on a number of examples.

The project report contains a summary of the approach used and the results achieved. Resilient systems are clearly necessary and relevant, and the outcomes of this project could serve as the technological foundation for building such systems.

2.0 INTRODUCTION

Software is likely to have latent defects and systems can break down because software breaks. The defects in software come to light because something changes – in the system or in its environment that it was not prepared for. Yet, we increasingly depend on software to build complex systems, including cyber-physical systems (e.g. vehicles, machines, and infrastructure) where software defects are always unacceptable, and often catastrophic. Hence software-integrated systems need to be more resilient to changes in the environment and the system, as well as faults. Our goal with the project was to provide an answer to the question: "How do we construct systems that are resilient to changes, including faults in the system and radical changes in the environment?"

Fault tolerance in computing has a long history, but resilience is beyond the capabilities of current fault-tolerant systems – as resilience means 'adapting to change'. The DoD OSD defines a resilient system as: "*A resilient system is trusted and effective out of the box in a wide range of contexts, easily adapted to many others through reconfiguration or replacement, with graceful and detectable degradation of function*."

In the project we developed an end-to-end comprehensive, model-based approach to the engineering of resilient software systems. We claim that to realize resilience in a system, several technologies are needed:

1. A domain-specific modeling language to support architectural modeling of resilient component-based systems
2. Design patterns for resilience to provide a library of reusable design solutions that can be applied to architecting resilient systems
3. Verification techniques to enable the verification of the system to achieve dependability
4. Software generators to automate the generation of application-specific code to support resilience functions
5. Middleware libraries to provide a library of reusable services for resilient systems

Our vision was to build technology and tools for architects and developers that assist them in creating resilient systems. The implications of such technology on current software development processes are profound: verified systems that are created with reduced effort yet are able to adapt to changes and faults, even without human intervention.

Our plan addressed the topics: 1, 2, and 4 of the above list, develop solutions for them, and to evaluate the solutions on realistic examples, on a small, distributed embedded computing platform. We have accomplished these goals, as described in the chapters below.

During the performance period, the tasks were extended with the following elements:

A. Develop system architecture trades for robustness and resiliency in spectrum challenged environments, particularly in missions associated with fractionated space architectures using strategies such as run time and temporal logic verification methods.

B. Using existing architectures and models from DOD programs, employ model checking at the hardware, network, and application layer to identify system vulnerabilities in various system configurations.

C. Using a variety of model checking frameworks including temporal logic, analytic, and probabilistic, determine the best mix of techniques to verify performance of distributed heterogeneous dynamic infrastructures such as distributed communications space architectures.

D. Understand the embedded computational architecture requirements needed for online systems verification in distributed heterogeneous information architectures.
E. Investigate cost tradeoffs in large system architectures such as space platforms between online model checking and fault repair as opposed to redundancy and platform shielding for hardware and software failure management.

Our plan was to address these questions by using and extending the results of the baseline tasks (1, 2, and 4 listed above), and create demonstrative examples to show the results.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The research method we followed was based on rapid development, prototyping, and evaluation of concepts and solutions. The project was focusing on the implementation, testing, and evaluation on small examples. Our aim was to develop the software engineering techniques and technology for building resilient software systems. Based on our prior experience and background, we have done this work following a model-driven engineering approach.

Model-driven development and engineering (MDE) [1][2] in general, and the particular approach we use in our research and tools, called model-integrated computing (MIC) [3] relies on the use of domain-specific modeling languages for creating models of the system to be built, analyze the systems based on those models, and then generate significant parts of the implementation of the system. Additionally, the models can also play an active role during run-time, for instance, they can provide a run-time representation of the architecture of the software system, hence giving reflective capabilities to it. We have followed the MIC method here: we designed a domain-specific modeling language for modeling resilient software architectures, we have developed tools that can verify the system based on the models, and we have developed software generators that generate code from the model to be compiled and the used in the running system.

Resilience [4] is a system level property – by definition any part of the system can fail, yet the system can be resilient and recover from failures. Hence our main assumption was that we need to work towards a solution that makes a system resilient, not just one more of its components. Hence, our products address system level issues, including architectural solutions to resilience. The modeling language, the analysis, and the run-time tools created focus on the entire software architecture, not only on individual components, or elements.

Another assumption was that the software system we are building is component based. In other words, the software system is constructed from software components that can be individually started and operated, and all the functions of the system are provided by the active running components. We have used a fairly generic component model, but our experiments were relying on using the Linux operating system, running software components loaded into the processes. We believe the results can be generalized to other platforms as well.

4.0 RESULTS AND DISCUSSION

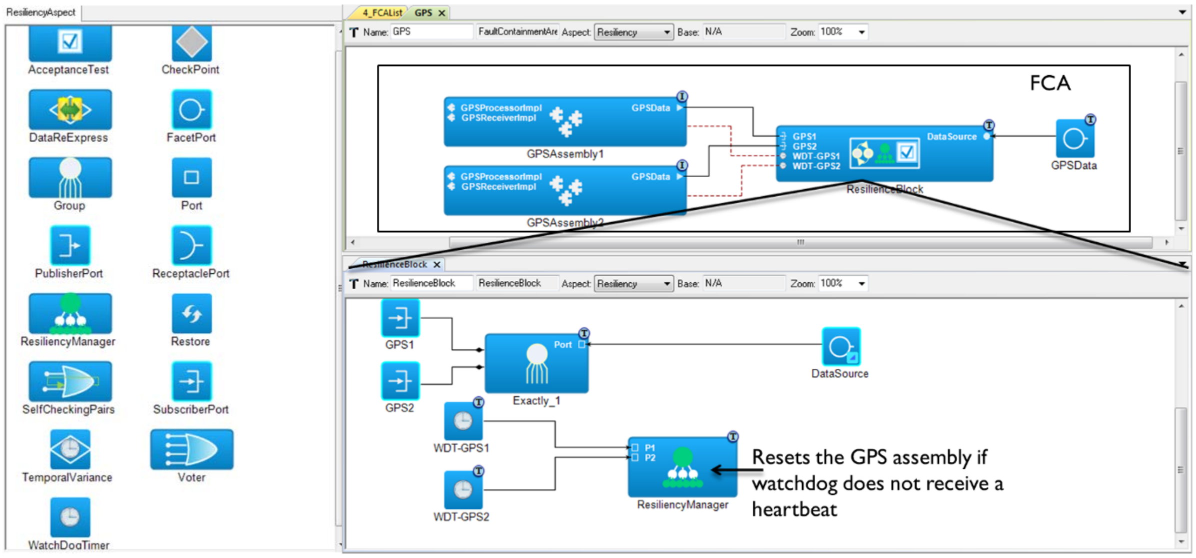
The main results of the research are described in the subsequent sections.

4.1 Domain-specific modeling language

The modeling of resilient software systems is beyond the capabilities of current architecture modeling approaches, including AADL or UML. Although it is understood that UML is extensible through the profiles and stereotyping mechanism, to our knowledge, no work has been done to specifically address resilience in a UML profile.

The starting point for the modeling language was our previous work distributed real-time embedded systems that resulted in a software platform and model-driven development toolchain called DREMS (for Distributed Real-time Embedded Managed Systems)[5]. We have significantly revised and extended that modeling language to support resilience features. The new language is called ReSoSML, for Resilient Software Systems Modeling Language.

In the course of the project we have created two versions of ReSoSML: the first version supported explicit specification of redundancy and resilience features. The supported architectural patterns included: self-checking pair, triple-modular redundancy (TMR), checkpoint/restore, watchdog timer, data re-expression, replication, and others. The figure below shows a subset of the graphical elements and some illustrative models for these patterns.



Resets the GPS assembly if watchdog does not receive a heartbeat

Figure 1: Resilience blocks and example model

In the course of the evaluation of this first version of the modeling language, we learned that it can be quite complex to use, as the modeler has to be very specific about all the details of the resilience techniques to be used. Furthermore, the models were focusing on the faults in the specific software components, and it was not possible to express what the system should do if a hardware node fails. Finally, the solutions were not connected to the functions of the system (i.e. the services the system should provide through the software applications), rather to the handling (detection and mitigation) of failures in individual software components.

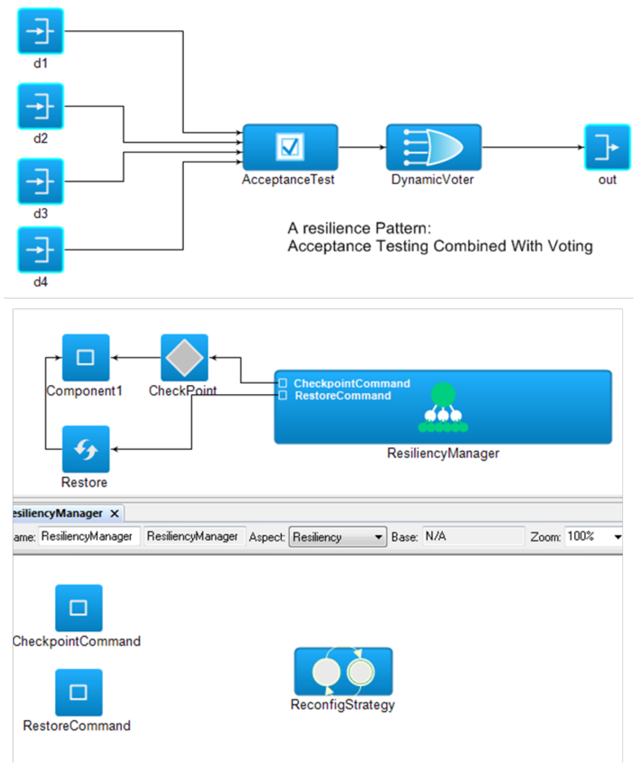
This led us to the design of the second version of the language that is much simpler, yet more powerful. The key insight in the new version is that a very generic resilience modeling is possible if one uses a symbolic and implicit specification of *what* the system should do and *how* it can do that. We have achieved this in two steps:

(1) We explicitly modeled the system functions and their mapping to application software components. The system functions modeled followed the standard functional decomposition (based on systems engineering standards, like MIL-STD-499) with the leaf functions being mapped to software components or combinations of software components. The combination operators included AND and M-of-N.

(2) We modeled the allowed and disallowed mapping of components to hardware resources via logical relationships over sets (expressed as constraints). These models implicitly represent all possible system configurations.

4.2 Design patterns and resilience scenarios

In the first generation of the modeling language we have built support for the following design patterns: Voter, Acceptance test, Self-checking pair, Checkpoint/restore, Data re-expression block, Temporal variance block, and Watchdog timer. These elements could be used in a 'Resilience Block' – a special kind of software component that captures fault detection and mitigation logic. The figure below illustrates the use of the resilience blocks.
The support for the design patterns was implemented in the form of explicit modeling constructs that can be reused as resilience patterns. Figure 2 shows an illustrative model.



ame: ResiliencyManager ResiliencyManager Aspect: Resiliency Base: N/A Zoom: 100%

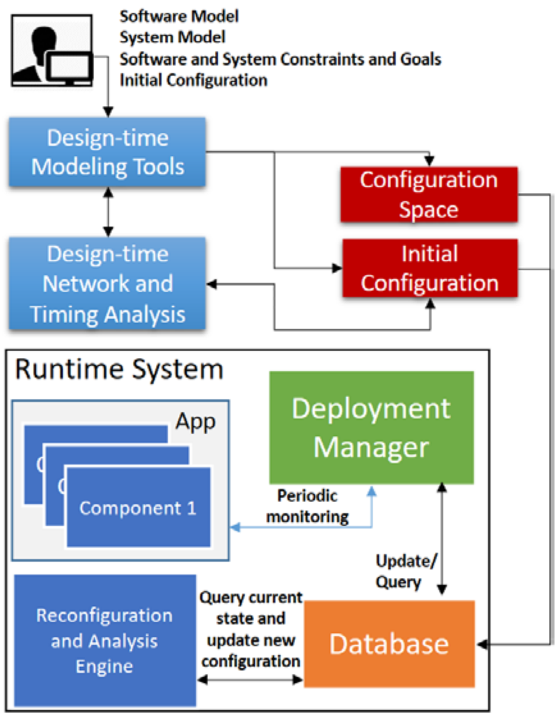
Figure 2: Example model for using the resilience block

The modeling approach has been tested on small scale examples but it was somewhat difficult to use. Modelers had to remember the various resilience design patterns and how they were to be used. When a design pattern was used, the number of elements would have to change depending on the specific applications. For instance, a design pattern like 'Acceptance Testing Combined with Voting' would have to be adjusted to the 3 channel (i.e. triplex), or the 4 channel (i.e. quadruplex) cases – a rather cumbersome approach.
Analyzing this situation we concluded that we need a better approach for modeling resilience. For that, we need to learn how resilience can be implemented in actual systems. As a consequence, we have (1) developed a documentation template to describe resilience scenarios, and (2) we have documented a number of scenarios that we identified from our experience and from the literature. The description template is illustrated on Table 1 below.

Table 1: Description template for resilience scenarios

Name	*Short name of the scenario*
Description	*Detailed, narrative description of the scenario*
Location	*Location of the issue initiating the scenario (within a system)*
Issue	*Description of the issue (i.e. the root cause)*
Anomaly	*Anomaly (deviation) locally caused by the issue*
Effect	*Observable functional effect caused by the issue*
Impact	*System-level impact of the issue*
Detection	*Location and process for the detection of the anomaly or effect*
Diagnostics	*Location and process for the diagnostics (i.e. cause identification)*
Mitigation	*Location and process for the mitigation of the effect of the issue*
Recovery	*Location and process for the recovery from the issue*

The template allowed us to develop and document a number of resilience scenarios that could arise in a component-based software system. These scenarios are documented in Appendix B. The conclusion of this activity was twofold:



Software Model
System Model
Software and System Constraints and Goals
Initial Configuration
Design-time Modeling Tools
Configuration Space
Initial Configuration
Design-time Network and Timing Analysis
Runtime System
App
Component 1
Deployment Manager
Periodic monitoring
Update/ Query
Reconfiguration and Analysis Engine
Query current state and update new configuration
Database

Figure 3: Resilient run-time system architecture (with design tools)

(1) Fault tolerance techniques for software are well-developed and documented [6] [7], but expressing them as a design pattern supported by a modeling tool is somewhat problematic. It is important for a software engineer to know these techniques, but their graphical representation is somewhat hard to use.

(2) Implementation of support for resilience in a system is best understood through examples. We found the description template (described above) very useful, as it allows us to document and analyze quite a number of scenarios.

As a next step, we devised a general approach to building resilient systems using model-based techniques. The approach relies on a generic architecture shown on Figure 3. The top of the figure shows the model-based design tools that developers use to create system architectural models that capture an *initial* configuration of the system, as well as a potential configuration *space*. A configuration is defined as the set of all possible system configurations, i.e. system functions mapped

to software components that are deployed on hardware resources.
These models are utilized in a run-time system, shown at the bottom. The software applications are built from components, and they are configured using a Deployment Manager (DM) – a system service that launches and configures the (distributed) applications. This service is also responsible for monitoring the running applications, as well as monitoring other deployment managers running on the other nodes of the network (not shown on the figure). When a fault is detected in an application or in another host node, a Reconfiguration and Analysis Engine (RAE) is activated that computes a new configuration for the system and instructs the Deployment Manager to perform a reconfiguration. The communication and interaction between these two large components happens via a shared distributed, fault-tolerant database. The details of this mechanism are described in a publication [P6].
One of the key concepts in this architecture is the way the system is modeled in terms of functions, hardware resources, software components, and deployments. The UML class diagram on Figure 4 shows a conceptualization.

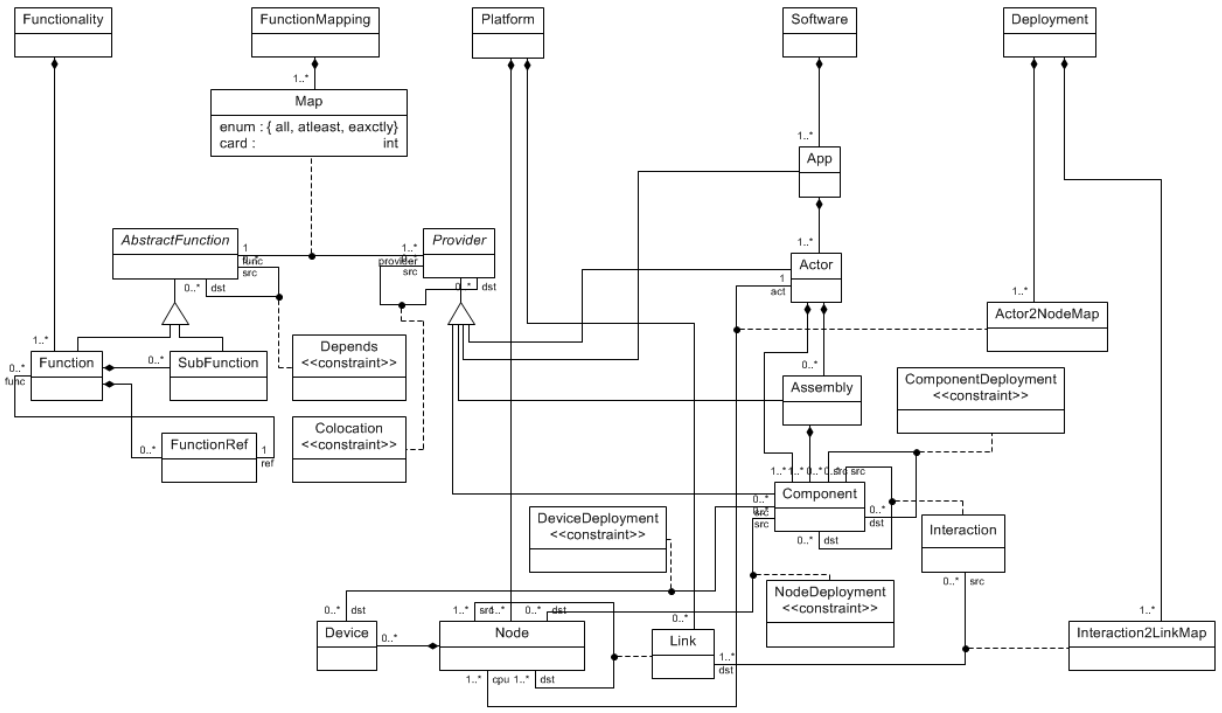


Figure 4: Resilience model conceptualization

Reading the diagram from left to right, first, we model a functional decomposition of the system, in terms of functions and their sub-functions. These functions are eventually mapped to specific software components, component assemblies, actors (processes), or applications. This mapping can be one-to-one, or one-to-many with the qualifiers ‘all’, ‘at least’, ‘exactly’; indicating how many and which of those function providers are needed. The hardware platform is modeled as computing nodes that can have attached hardware devices, and are linked through network links. Software is modeled as a hierarchy of applications containing actors containing assemblies containing components. Components interact with each other through various interaction patterns: publish/subscribe and/or client/server. Finally, actors and interactions are mapped to computing nodes and communication links, respectively.

This modeling approach allows the description of a specific software architecture mapped onto a specific hardware architecture, but if we allow alternatives in the mappings it can also represent a *configuration space* over the architecture.

In the run-time system not only the binary executables of the application (components) are used, but also a compact representation of the software architecture, specifically: the initial configuration and the configuration space. This data structure is commonly referred to as the 'resilient data model (RDM) for the deployment plan' [P5]. The key observation here is that system resilience can be implemented with the help of flexibility in the deployment. If a software component, application, or hardware node or communication link fails, then the software application(s) can be restarted and/or deployed on the hardware platform in a different configuration, such that the required system functions are still provided. In other words, the services are still provided, although via a different arrangement of hardware and software resources.

The technical solution to representing a configuration space (as opposed to representing a single configuration) is based on an encoding of the space as a constraint logic programming problem (CLP) [P6]. The encoding represents the RDM as a set of constraints over integer variables, where a valuation of the variables represents a particular configuration of the system. Configuration choices and requirements, like 'component X can be mapped to nodes A and B' are also represented as Boolean expressions, like, e.g., $x = a \lor x = b$. The constraint-based representation can encode a potentially very large configuration space, as it does not encode configurations individually, rather it encodes them in an implicit, symbolic form. System functions are also represented by variables and their mapping to software components is represented via relations. In other words, the entire system is represented as complex CLP problem.

The CLP representation can be solved, i.e. a specific configuration can be computed by using a constraint solver. Similarly, if a solution is already known, (i.e. the valuation of variables is given), then it can be validated by the evaluating the constraint expressions; this should yield a 'true' value for all expressions. The CLP representation also allows modeling the effect of faults. If a hardware node fails, a new constraint is added to the constraint system, representing the fact that no software component can be allocated to that node. Re-running the solver, a new configuration will be computed, that bypasses the failed component.

In the run-time architecture, a key component is the 'resilience and analysis engine' (RAE) that, in conjunction with the deployment manager (DM) is responsible for managing the software configuration and providing resilience. The RAE hosts the constraint solver discussed above. The attached database (DB) stores the RDM in a persistent, distributed, and fault-tolerant manner. The DB is replicated on each hardware node of the network, and whenever it is updated on any node, the changes are propagated to all the replicas. (We rely on the MongoDB tool [9] to achieve this.).

Initially the DB loaded with the RDM that includes the initial configuration of the system. The DM detects this change and communicates with the RAE to perform the validation of the initial configuration. The RAE invokes the constraint solver that validates that the configuration (i.e. a valuation of the encoding's variables) satisfies the constraints of the system. Once this validation passes the DM deploys and activates the software components.

Whenever a fault is detected, the RAE receives a notification about the nature of the fault. The RAE then inserts a corresponding new constraint into the constraint set and re-runs the solver that computes a new configuration. This is deposited in the DB, and then the DM is notified that should modify the current configuration of the software components to reflect the necessary change. Note that this scheme will find a new configuration if there is one, and if there is none it is able to report it – due to the fact that the configuration is a solution to the CLP problem. However, it is acknowledged that the computation of a new solution may take a relatively long time due the nature of the constraint solving algorithms (which are complex search algorithms).
To summarize, our approach to resilience is as follows:

1. Represent the configuration space of the software system as a CLP problem. The configuration space is the set of all possible deployments of the software components on hardware and communication resources. One point in the configuration space is the initial configuration.
2. At run-time, use an on-line constraint solver engine to validate and compute new configurations for the software, if something fails in the system. Upon a fault in the system, a new constraint is added to the CLP problem and a new solution is computed which does not include the failed component. Then the system is reconfigured accordingly.

4.3 Software generators for middleware

In the course of the project a new software generator has been developed for generating the RDM and we have reused existing generators from the ACE/TAO/CIAO/DANCE software infrastructure [8]. The reused software generators were updated according to the needs of the project. The most significant change was in the DANCE (Deployment And Configuration Engine) that was modified to act as the DM and interfaced to the DB.
The developed software generator is a model interpreter (embedded in the Generic Modeling Environment that has been used as the platform for implementing the modeling tool) that traverses the models and populates the DB according to the RDM schema. As the modeling language is already similar to the RDM schema, this mapping is accomplished by a rather straightforward algorithm whose details are described in the publication [P5].

4.4 Resilience metrics and calculations

The approach to resilience described above allows not only the operational implementation of resilience, but also the evaluation of architectures with respect to resilience. Our previous work on an Information Architecture Platform for fractionated satellites has taught us the importance of understanding a system’s capabilities with respect to faults in the system. In a fractionated space system functionality is distributed across satellite nodes that run software applications. Each node may have a dedicated hardware resources (e.g. sensors, compute engines, on-board storage) and the flexibility of the system comes from the feature that software applications can be deployed in various ways, using various resources to provide the services needed. Given a hardware platform (compute nodes, network links, special devices attached to nodes), the question arises: how resilient is a particular hardware/software architecture to faults in any of the components? Note that the system’s flexibility comes from the multitude of potential deployments of the software.
In order to evaluate architectures with respect to resilience we have developed a resilience metric [P3] that is computed over the (1) hardware configuration, (2) software application architecture,

(3) deployment constraints (that restrict how the software components can be deployed on the hardware resources), and (4) an initial configuration (i.e. deployment). The metric is defined as a pair of integers: $R = (r_{min}, r_{max})$, where $r_{min}$is defined as the *least* number of failures that the system will tolerate without any impact on the mission, and r_{max} is defined as the *maximum* number of failures that can be sustained while supporting the mission remains feasible. The first number is also called the worst-case resilience, while the second as best-case resilience. In other words, (1) the system will always tolerate r_{min} failures, but with $r_{min} + 1$ the system *can* fail, and (2) the system can tolerate up to r_{max} failures but $r_{max} + 1$ *will* make the system fail. Between $r_{min}$and r_{max} the system may fail (but this is not certain).

This metric allows the comparison of alternative architectures (i.e. hardware and software configurations and deployments). It is interesting to note that even for simple systems the size of configuration space can be quite large. One example system that was based on a fractionated satellite problem had a space of size 10^5, while the resilience metric was $R = (1,17)$ (indicating that the system could tolerate 1 failure in the worst case and 17 in the best case).

The calculation of the resilience metric is not trivial, and we have developed an algorithm that relies on a CLP solver (more specifically, on an SMT solver). The algorithm encodes the problem as a CLP problem, exhaustively introduces faults in components and asks the solver to compute solutions (i.e. configurations). This basic approach is optimized via using a compact and efficient encoding of the problem. While the algorithm is clearly not an efficient one, it can compute the metrics with reasonable performance. For instance, for the problem mentioned above that completed the work in a few minutes on an average desktop machine. A point to note is that we have used a very efficient SMT solver: the Z3 system from Microsoft Research [10].

4.5 Verification of network performance

In systems where communication resources are constrained design-time verification of communication performance is important. Based on our earlier work on fractionated satellite systems, we have developed an approach to accomplish this. The approach is described in detail in [11], here we give a summary.

The approach is reliant upon a model of the network performance (as a function of time) and a model of the application requirements (as a function of time). The network performance is modeled as cumulative function – called the *network profile* – describes the amount of data that *can* be transferred through a network up to a point in time *t*. The application requirements are defined for each communication port of each component in the application. This is another cumulative function – called the *application profile* — that represents the amount of data that *will* be produced by that port up to a point in time *t.* Now using techniques, similar to the ones developed in Network Calculus (NC) [12] the network capabilities and the (combined) application requirements can be composed and the overall system performance validated. Our approach is slightly different from NC as it operates with cumulative functions, not data rate functions, yielding more accurate results. The mathematical foundation of the composition is the '(min,+)' calculus (used to substitute the conventional (+,*) operators) applied in a convolution operation. The network profile is convolved with the summed-up application profiles using the (min,+) rule, yielding the worst case latency and buffering requirements in the network.

The approach works well for systems where the network profile is a periodic function (in the rates). This is typical in fractionated satellites due to the orbital mechanics: bandwidth between satellites tends to fluctuate as a sinusoidal function of time.

The results of the calculation verify that the network is capable of handling the application requirements. If the network buffer requirements (and/or delays) grow unboundedly over time, then the network is not capable of handling the load. If they remain within bounds, then we know what worst case network latency and buffer utilization we can expect from the system. Figure 5 shows an example. The system (network) provided profile is p[t], the application required profile is r[t] and the resulting actual profile (data delivered as a function of time) is shown as l[t]. The maximum vertical difference between r[t] and l[t] gives the worst case buffer size needed, and the maximum horizontal distance shows the worst-case delay.

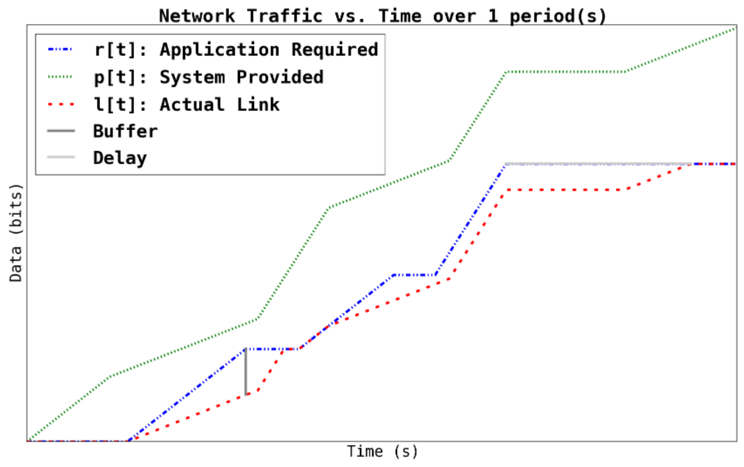


Figure 5: Example results for network performance calculations

The analysis results can be used as an admittance test for application deployments, at design-time. Additionally, as the calculations are rather simple, they can be performed at run-time, on the newly computed configurations. The paper [P6] describes such a run-time approach, where the solver-generated new architecture is validated with respect to network performance, prior to deployment. If the validation fails (i.e. the new configuration would overload the network), then the solver is forced to compute a new solution. This will be subjected validation before deployment. The process may iterate until an acceptable solution is found or no more solutions are available.

4.6 Computational requirements for online verification

The resilience approach described above is clearly reliant on the performance of the RAE. Due to the encoding of the problem as a CLP problem, we have to rely on a CLP solver to compute new solutions. This solver acts as an online verification engine that computes new configurations at run-time. As the computation is based on strict constraints that the solution has to satisfy, the computed configurations are always valid.

The main computational load for reconfiguration comes from the solver that implements a search algorithm. Note that all the other components in the run-time system (i.e. DM and DB) are either deterministic or could be made deterministic. The mapping of the RDM into the encoding of the problem as a CLP is deterministic and is bounded by the size of the RDM (nodes and edges in the model). The key performance issue is in the solver that has to compute a valuation for the solution variables.

As the solver is a very complex, possibly adaptive algorithm, it is not possible to come up with a closed formula for its performance. In our example system we have used the best known and available solver: Z3 from Microsoft Research, an open source tool. This tool uses problem rewriting, machine learning, multiple solver strategies and heuristics in an overall adaptive framework that delivers best-of-breed performance. In our, admittedly small scale experiments the run-time computation of a solution was accomplished in a few seconds (on an average desktop), and that performance, given the overall speed of reconfiguration of the system, was acceptable.

To get better, more deterministic behaviors from the solver design-time pre-computation of the solutions is possible. However, caching the solutions for run-time use may not be feasible, as the configuration space could be quite large and on-board storage can be limited. However a design-time analysis and testing of the solver's performance may be feasible. How to optimize the search for a given architecture and system could be a very interesting and relevant area of future research.

4.7 Tradeoffs between online verification and conventional solutions

Resilience can be supported by conventional methods, like classic fault tolerance via redundancy and very simple failover logic or even static fault protection methods like shielding, or via the more complex, dynamic scheme that we have proposed and prototyped that relies on online verification and reasoning. It is important to understand the differences and cost/benefit factors for both approaches. Our findings are summarized in a report (attached in Appendix C), here we give only a brief summary.

Shielding of sensitive electronics to achieve resilience against radiation effects is very expensive in terms of weight and size. The shielding required depends on the orbit's position situation (LEO vs GEO). Shielding is simple to implement and it can be low risk to install as it does not interact with anything in a system (except the geometry and the weight). However shielding cannot completely isolate the electronics from radiation effects due the function of the electronics (e.g. sensors), but it can reduce the impact of radiation. Nevertheless, a well-designed system should have functional (dynamic) means to provide resilience. Arguably, these functional methods induce less weight and size penalties.

Conventional redundancy is well-established, but it is also expensive in terms of the cost of the electronics itself, and the resulting weight and power requirements. While it is relatively simple to implement, some elements may have to be shielded or be made highly radiation tolerant (e.g. the voter in a TMR). It is medium risk to implement, but it certainly introduce additional complexity in the system.

The online verification based approach described in this report is the most complex, and consequently, the most high-risk approach. Its physical size, weight, and power impact are negligible (it is implemented in software), and if there is a way to recover for the system, it is able to find it. While the physical properties of the approach are appealing, the additional complexity is a risk. The RAE described above is quite complex and includes elements that are hard to test or verify (e.g. the constraint solver). Using it may introduce complexity-related risks into projects, so it should be used only if benefits (higher flexibility, less redundancy required, etc.) clearly overtake the risks.

4.8 Demonstration system

We have built a prototype demonstration system that includes a modeling tool, software generators, design-time analysis tool for resilience metric calculations and network performance calculations, and a run-time resilience analysis engine. The demo system includes a prototype of

all architectural elements from Figure 3. It requires a Windows machine to run the modeling and analysis tools and Windows or Linux machines (or virtual machines) to run the reconfiguration engine.
The demonstration system is available from the website: https://phab.resos.isis.vanderbilt.edu.The system requires a GME installation (available from https://www.isis.vanderbilt.edu) and it comes with its own installer.
The system includes a few models that demonstrate: (1) the modeling approach using the domain-specific modeling language, (2) the analysis of the models with respect to resilience metrics, and (3) the execution of a reconfiguration in a run-time system. For the latter, a Z3 solver and a MongoDB setup is also required.
The example uses 3 satellites (compute nodes) linked via a wireless network. The first satellite called Alpha has a high- and a low-resolution camera and a GPU (for processing), the second satellite Beta has only a GPU, and the third satellite Gamma has a high-resolution camera. All satellites have their own satellite bus, ground link, and wireless network link (to communicate within cluster).
There are three software applications, two related to flight controls, and another one doing wide area monitoring. The two flight control applications are the (1) `ClusterFlightApp` that is responsible for maintaining the coordinated cluster flight and the (2) `SatelliteFlightApp` that maintains the orbit of one satellite but directly controlling the satellite bus. The `ClusterFlightApp` exists in a single copy on the cluster and it requires the ground communication link. The SatelliteFightApp requires the interface to the satellite bus and it has to be located on the same satellite (i.e. it cannot control the bus of another satellite). The wide area monitoring application requires access to the high- and low-resolution cameras, and the GPU.
All of the above specifications have been captured in the models, from which an RDM can be instantiated and then the system encoded as a CLP problem.
The figures below show an initial configuration of the system, the situation when a camera fails, and the result of the reconfiguration. All the drawings were generated from the running application on a simulated cluster.

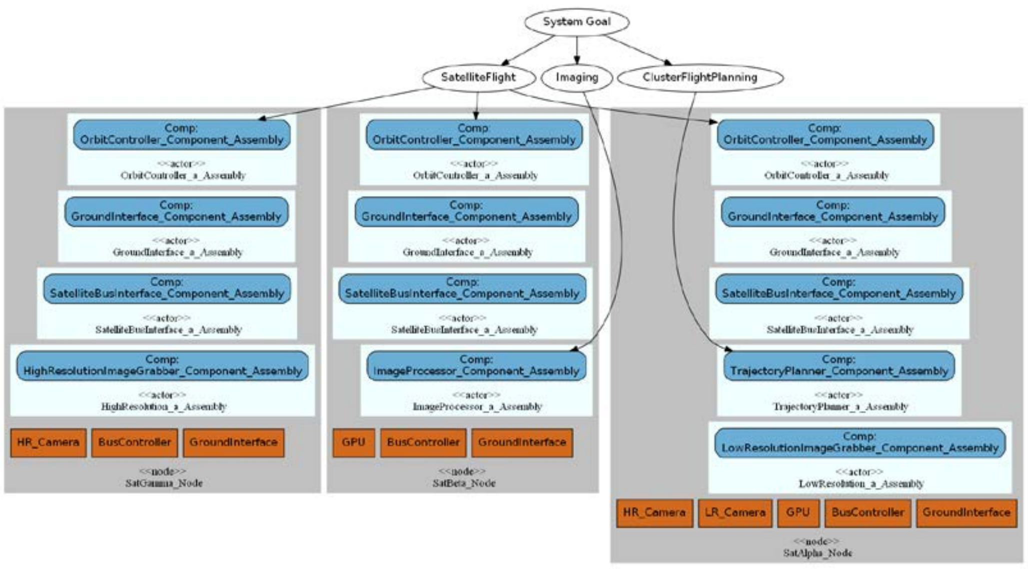

System Goal
SatelliteFlight
Imaging
ClusterFlightPlanning
Comp: OrbitController_Component_Assembly
<<actor>> OrbitController_a_Assembly
Comp: GroundInterface_Component_Assembly
<<actor>> GroundInterface_a_Assembly
Comp: SatelliteBusInterface_Component_Assembly
<<actor>> SatelliteBusInterface_a_Assembly
Comp: HighResolutionImageGrabber_Component_Assembly
<<actor>> HighResolution_a_Assembly
HR_Camera
BusController
GroundInterface
<<node>> SatGamma_Node
Comp: ImageProcessor_Component_Assembly
<<actor>> ImageProcessor_a_Assembly
GPU
<<node>> SatBeta_Node
Comp: TrajectoryPlanner_Component_Assembly
<<actor>> TrajectoryPlanner_a_Assembly
Comp: LowResolutionImageGrabber_Component_Assembly
<<actor>> LowResolution_a_Assembly
LR_Camera
<<node>> SatAlpha_Node

Figure 6: Initial configuration of the satellite example

The initial configuration shows the required system functions on the top, and their mapping to software components that are allocated to the hardware nodes.
When the GPU fails on satellite Beta the ImageProcessor (that is the main component of the wide area monitoring application) cannot function, as it does not have access to the specialized

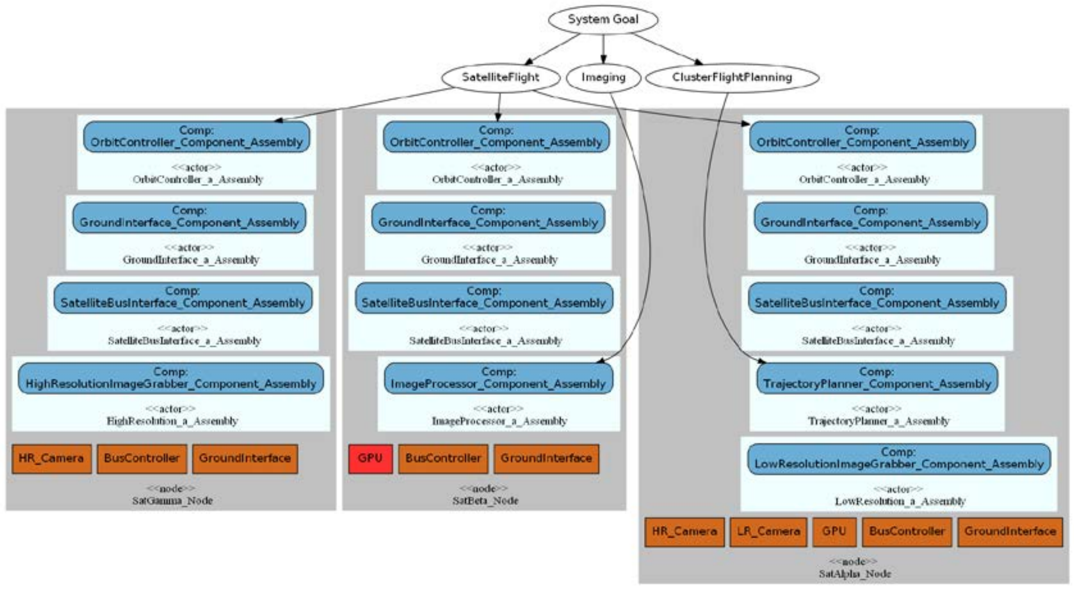

System Goal
SatelliteFlight
Imaging
ClusterFlightPlanning
Comp: OrbitController_Component_Assembly
<<actor>> OrbitController_a_Assembly
Comp: GroundInterface_Component_Assembly
<<actor>> GroundInterface_a_Assembly
Comp: SatelliteBusInterface_Component_Assembly
<<actor>> SatelliteBusInterface_a_Assembly
Comp: HighResolutionImageGrabber_Component_Assembly
<<actor>> HighResolution_a_Assembly
Comp: ImageProcessor_Component_Assembly
<<actor>> ImageProcessor_a_Assembly
Comp: TrajectoryPlanner_Component_Assembly
<<actor>> TrajectoryPlanner_a_Assembly
Comp: LowResolutionImageGrabber_Component_Assembly
<<actor>> LowResolution_a_Assembly
HR_Camera
LR_Camera
GPU
BusController
GroundInterface
<<node>> SatGamma_Node
<<node>> SatBeta_Node
<<node>> SatAlpha_Node

Figure 7: Fault in the GPU on Satellite Beta

image processor. Hence a reconfiguration is initiated that results in moving the application to satellite Gamma that has a functional GPU.

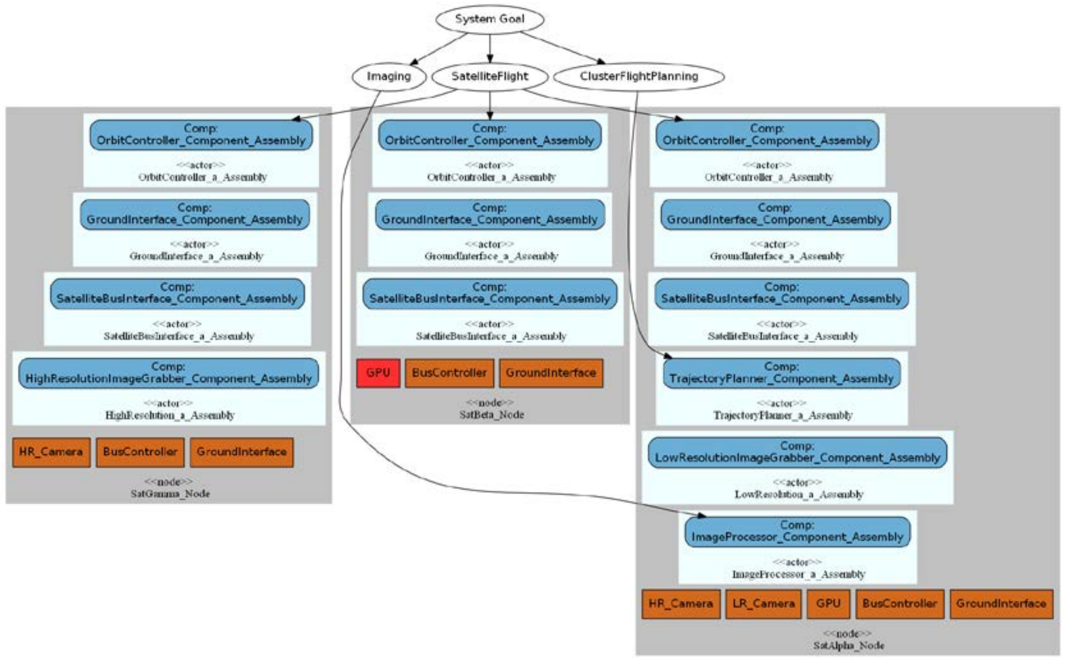

System Goal
Imaging
SatelliteFlight
ClusterFlightPlanning
Comp: OrbitController_Component_Assembly
<<actor>> OrbitController_a_Assembly
Comp: GroundInterface_Component_Assembly
<<actor>> GroundInterface_a_Assembly
Comp: SatelliteBusInterface_Component_Assembly
<<actor>> SatelliteBusInterface_a_Assembly
Comp: HighResolutionImageGrabber_Component_Assembly
<<actor>> HighResolution_a_Assembly
Comp: TrajectoryPlanner_Component_Assembly
<<actor>> TrajectoryPlanner_a_Assembly
Comp: LowResolutionImageGrabber_Component_Assembly
<<actor>> LowResolution_a_Assembly
Comp: ImageProcessor_Component_Assembly
<<actor>> ImageProcessor_a_Assembly
HR_Camera
LR_Camera
GPU
BusController
GroundInterface
<<node>> SatGamma_Node
<<node>> SatBeta_Node
<<node>> SatAlpha_Node

Figure 8: Result after reconfiguration

5.0 CONCLUSIONS

In the course of this project we have developed an approach to modeling resilient software systems and a concrete approach to resilience in component-based software systems. It is our conclusion that model-based development and engineering is necessary for such systems, due to the inherent potential complexity of these systems.

We have developed two techniques for modeling resilient software architectures: one based on conventional patterns, and another one based on a more flexible and general approach. While both of them were useful, the first one suffered from usability problems, while the second one was more generic and less complex.

We have analyzed various resilience scenarios using a template we developed. The template allows the documentation of scenarios and can assists a system architect in developing solutions.

We have developed an approach to facilitate run-time resilience through a resilience engine. The method encodes the configuration space of the system in a mathematical model and then uses a general purpose constraint solver to compute solutions that are alternative configurations of the system that can have failing components. The approach has been prototyped in a demonstration package.

Our conclusion is that a flexible, dynamic, and adaptive computational solution to resilience is feasible. However it should be applied with caution as it significantly increases the complexity of the system and makes testing and verification problematic.

6.0 REFERENCES

[1] Soley, Richard. "Model driven architecture." OMG white paper 308 (2000): 308.

[2] Kleppe, Anneke G., Jos B. Warmer, and Wim Bast. MDA explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003.

[3] Sztipanovits, Janos, and Gabor Karsai. "Model-integrated computing." Computer 30, no. 4 (1997): 110-111.

[4] Laprie, Jean-Claude. "From dependability to resilience." In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks, pp. G8-G9. 2008.

[5] Levendovszky, T.; Dubey, A.; Otte, W.R.; Balasubramanian, D.; Coglio, A.; Nyako, S.; Emfinger, W.; Kumar, P.; Gokhale, A.; Karsai, G., "Distributed Real-Time Managed Systems: A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems," Software, IEEE , vol.31, no.2, pp.62,69, Mar.-Apr. 2014 doi: 10.1109/MS.2013.143

[6] Hanmer, Robert. Patterns for fault tolerant software. John Wiley & Sons, 2013.

[7] Knight, John. Fundamentals of Dependable Computing for Software Engineers. CRC Press, 2012.

[8] ACE/TAO/CIAO/DANCE middleware software infrastructure, available from http://www.theaceorb.nl/en/ace-tao-ciao-dance-prod

[9] MongoDB database http://www.mongodb.org/

[10] Z3 high-performance solver http://z3.codeplex.com/

[11] Emfinger, William, Gabor Karsai, Abhishek Dubey, and Aniruddha Gokhale. "Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks." In Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems, pp. 44-47. ACM, 2014.

[12] Le Boudec, Jean-Yves, and Patrick Thiran, eds. Network calculus: a theory of deterministic queuing systems for the internet. Vol. 2050. Springer Science & Business Media, 2001.

APPENDIX A: PUBLICATIONS

[P1] Pradhan, S.; Emfinger, W.; Dubey, A.; Otte, W.R.; Balasubramanian, D.; Gokhale, A.; Karsai, G.; Coglio, A., "Establishing Secure Interactions across Distributed Applications in Satellite Clusters," *Space Mission Challenges for Information Technology (SMC-IT), 2014 IEEE International Conference on* , vol., no., pp.67,74, 24-26 Sept. 2014, doi: 10.1109/SMC-IT.2014.17

[P2] Balasubramanian, D.; Dubey, A.; Otte, W.; Emfinger, W.; Kumar, P.; Karsai, G., "A Rapid Testing Framework for a Mobile Cloud," *Rapid System Prototyping (RSP), 2014 25th IEEE International Symposium on* , vol., no., pp.128,134, 16-17 Oct. 2014, doi: 10.1109/RSP.2014.6966903

[P3] Balasubramanian, D; “Quantifying Resilience in Component-Based Software Architecture Models”, at S5- Safe & Secure Systems and Software Symposium, 2014

[P4] Balasubramanian, D., Levendovszky, T., Dubey, A., & Karsai, G. (2014). Taming Multi-Paradigm Integration in a Software Architecture Description Language. In *8th International Workshop on Multi-Paradigm Modeling MPM 2014* (p. 67).

[P5] Pradhan, S.; Otte, W.; Dubey, A.; Gokhale, A; and Karsai, G.: Key Considerations for a Resilient and Autonomous Deployment and Configuration Infrastructure for Cyber-Physical Systems. In:: IEEE. : 11th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe-2014)., 2014

[P6] Emfinger, W;, Kumar, P; , Dubey, A;, & Karsai, G;, “Towards Assurances in Self-Adaptive, Dynamic, Distributed Real-time Embedded Systems”, submitted to *Software Engineering for Self-Adaptive Systems III, Lecture Notes in Computer Science*

APPENDIX B: RESILIENCE SCENARIOS

Resilience Scenarios for Managed Distributed Real-time Systems

Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, 37212.

B-1. Introduction

Distributed real-time embedded systems that interact with the physical world are ubiquitous and pervasive. We are relying on an increasing number of such systems that provide services to a large number of users. Lately, a new subclass of these systems called Distributed Real-time Managed Systems (DREMS) [B-1] has emerged that is characterized by (a) requirements for sharing of computational resources between application provided by different organizations, (b) ability to support multiple, simultaneous missions operating at different time scales, (c) cloud of computing nodes with mobile ad hoc networking and dynamic group membership, and (d) integrated physical constraints and mobility that impacts the software application running on the system.

Owing to the mobile, distributed and remote nature of the deployed DREMS platforms, resilience is an important property. A resilient system is expected to provide persistent and justifiably trustworthy service when subjected to functional, environmental or technical changes. The changes themselves could be expected or unexpected, and could be persistent, transient or intermittent with short, medium or long term effects.

In this report, we use a specific implementation of the DREMS architecture described in [B-1]. The architecture consists of a design-time tool suite for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components and a run-time software platform for deploying, executing, and managing application software on a network of mobile nodes. The run-time software platform consists of an operating system kernel, system services and middleware libraries, where each layer is built upon the guarantees provided by the lower layer.

The outline of the report is as follows. Section 2 present related research and background on resilient systems. Section 3 presents brief overview of layered reference architecture for a system, with the guarantees and the assumptions of each layer. We describe the different services of the platform in Section 3.2. Thereafter, in the following sections we describe the resilience scenarios.

B-2. Resilient Software Systems

One of the necessary but not sufficient requirements for building a resilient system is fault-tolerance. In [B-2], Avizienis described fault-tolerance as the attribute of a digital system that keeps the logic machine performing its set of specified tasks when it encounters various kinds of failures in its components.

Most of the earlier work on software and hardware fault-tolerance focused strongly on using redundant components and design diversity to tackle design-time faults [B-3][B-4]. In this approach, the computer system is partitioned into modules, with each module dealing with a specific sub-function. This federated partitioning provided for fault containment with a set of identical computing processors running software for several sub-functions. This approach improved the dependability of the computing system with more efficient use of redundant hardware. However, this still left the system vulnerable against the software faults and common mode failures.

B-2.1 Diversified Design

In a diversified design, several variants of the same software are used with an acceptance test employed at the end to compare the output from each of the variant. The rationale behind this approach is the expectation that component built differently will fail differently [B-5]. A diversified design has at least two variants plus a decider, which monitors the outputs of the variants. Three such strategies for software fault-tolerance are: Recovery Block approach [B-6][B-7], N-version programming approach[B-8], and the N self-checking programming approach[B-3][B-9].

The recovery block technique [B-6][B-7] uses the checkpoint and restart technique to recover from a fault. It uses multiple versions of the same software module as alternates. Upon failure of an acceptance test, the checkpoint state is used to restart the computation by using the next alternate version. In this approach, at least two versions of software module are required. If all alternates fail, the module issues an exception to the rest of the system.

In the N-version programming approach, multiple versions of same programs is executed in parallel, with a voter selecting the output most likely to be correct. This approach is different from the recovery-block approach in that it does not require an application dependent voter and that it needs at least three versions of the same module to work. However, the parallel execution necessitates the need to ensure input consistency. In 1990, Brilliant, Knight and Leveson published results from a large-scale experiment conducted in N-version programming [B-10]. In the experiments, they prepared twenty seven versions of a program at two different universities and then executed them one million times. The results of the experiment were intriguing. They noticed that different versions were found to be reliable when used individually with only a small number of failures. However, when they correlated these failures across different versions they discovered that the number of input cases when more than one version failed was significantly large. Upon post failure analysis, they further discovered that correlated failures arise from logically unrelated faults in different parts of the algorithms. They hypothesized that most of the faults resulted from the fundamental flaws in the algorithms that the programmers designed. Therefore, changing the development tools or methods to create new versions did not reduce the number of correlated failures in N-version software.

N-Self-checking approach [B-3][B-9] is a combination of Recovery block approach and N-version approach. This approach has two variations. The first variation uses acceptance test for each version as in the recovery block approach, however, the acceptance test for each version can be different. By executing these versions in parallel, this approach enables switching of output in case of errors instead of restarting from previous checkpoint. The other variant uses a comparison technique. It groups the variants into set of two, with a comparison unit forwarding the result to a selection unit only if the two versions in a set produce identical results. The selection logic then selects the outputs from different version similar to the N-version technique. The drawback of using this technique is the possibility of running into situation where both versions in a set produce identical wrong output.

B-2.2 Recovery-Oriented Computing

In [B-11], Brown and Patterson found that the heterogeneity and complexity involved in most large-scale service systems inherently leads to unforeseen failures. Therefore, in Recovery-Oriented Computing (ROC)[1] [B-11][B-12] they concentrate on reducing time to recover from faults and thus offer higher availability and aims to reduce total cost of ownership. ROC emphasizes on testing recovery systems and helps make recovery procedures a holistic part of the architecture rather than a patch or add-on that leads to extra complexity.

They have proposed six techniques [B-13] for recovery oriented computing.

1. Redundancy: Introduce redundancy to reduce the probability of a fault due to single points of failures. Note that this generally adds to the initial setup cost of the system. Nevertheless, they argue that by increasing the number of available resources as a safety margin, the mean time to recover are reduced that increases availability and hence reduce total cost of ownership.
2. Partitioning: They advocate the use of partitioning the system such that to ensure fault-containment and ease of fault-isolation.
3. Testing of recovery mechanisms: They advocate the presence fault insertion capability in live systems to test recovery process. This they argue helps in running availability benchmarks, which allows for reduction in time taken to deduct error.
4. Aid in Diagnosis of the cause of error: They advocate that a ROC system should contain sensors that help an operator determine the cause of a problem.
5. Logging of operator inputs to enable undo: In [B-14], Brown and Patterson have stated that a system should have the capability to rewind and replay a set of inputs given by a user to assist in repair. It is a system-wide undo. They have used this technique to build and undo email store [B-14]. However, since system behavior is not always deterministic, this undo technique can only work for a class of system and for that system, the cost of storing the undo trace would be enormous.
6. Orthogonal Mechanisms: Fox and Brewer state that one can reduce the likelihood of a complete failure and increase availability by using multiple versions for the same function that are independent from each other.

B-2.3 Resilience Patterns

R. Hamner [B-15] summarizes some of the common architectural and application patterns relevant towards creating fault tolerant software. The architectural patterns such as Redundancy, Recovery Blocks, Fault Observer, Maintenance Interface, Software Update, and Units of Mitigation capture the design strategies to instrument application patterns for Detection,

[1]http://roc.cs.berkeley.edu/

Mitigation, Recovery and Fault Treatment. As the name suggests, the Detection patterns: System Monitor, Voter, Acknowledgement, Heartbeat, Watchdog, Checksum, Routine Audits, Error Containment, etc. deal with detecting and diagnosing the errors. The mitigation patterns: Overload Toolboxes, Reassess Overload Decision, Deferrable work, Equitable Resource Allocation, Fresh Work before Stale, Queue for Resources, and Shed Load deal with mitigating the effect of the fault. The recovery patterns: Quarantine, Restart, Rollback, Roll-Forward, Checkpoint, Failover, and Limit retries are associated with recovering the system to its nominal state. The fault treatment patterns: Reintegration, Reproducible Error, Revise Procedure, Root Cause Analysis, and Small Patches deal with correcting the faulty portion of the software system.

B-2.4 Model Based Software Health Management – Lessons Learned

As part of a research effort on Model-Based Software Health Management (sponsored by NASA's Aviation safety program) we adapted a diagnosis scheme used for Systems Health Management and applied it towards diagnosis of a software component assembly. The real-time health management scheme involved a two-level software health management scheme: Component-level Health Manager (CLHM) provided localized and limited service for managing the health of individual software components and a System-Level Health Manager (SLHM) managed the health of the overall system. In the following paragraphs, we summarize the lessons learned as part of this exercise that can be applied towards building resilient systems.

- Effects of Local Mitigation: While local mitigation actions provide a quick local response to an anomaly, they could introduce new or modified failure cascades. For example, consider the case in which based on a certain contract violation, a local mitigation action stops a data publisher component. While it prevented the publication of bad data (that could have potentially violated the contracts), the lack of data published can lead to a problem on the consumer side, due to lack of data or data validity violations. It is important that higher-level or other related health managers are aware of the mitigation actions. This would allow other health managers to recognize these effects and act accordingly.
- Alarm Timing Issues: It is important that all anomalies detected are time stamped using the local node clock. Apart from the issue of time-synchronization between the nodes, it is also important to deal with the issues of alarms not arriving at the appropriate health managers in the order of their detection. This can be either due to the varying network latency or task preemption. A robust health management scheme could deal with these issues by using a variety of techniques including robust algorithms that account for delayed alarms, configurable policies for alarm aggregation with an appropriately sized moving windows, etc.
- Masking of Fault Effects: The health manager should be aware of the automated fault-tolerance schemes that are designed into the assembly of software components. The fault-tolerance schemes are inherently designed to stop the cascade of certain kind of faults. In order to build a resilient system, the health managers should be made aware of any discrepancy or fault information that is available through these fault-tolerance and fault-masking techniques (e.g. Voters, Active Replicas, etc.). This would allow the appropriate managers to check if appropriate recovery actions can be performed on the faulty software components or computing nodes.
- Intermittent or Transient Faults and Alarms: It is possible that the failure source or the

alarms associated with the anomalies are intermittent, i.e. they are observed in one period but not observed in another. This intermittent behavior can be caused by a partial masking effect, or intermittent behavior in the original fault source or it can be due to the mitigation actions. The health-management framework should be capable of handling such intermittent and transient problems.

- System Hysteresis: It is possible that the despite the mitigation action taken at the system-level to remove the fault source, the fault cascade remains in the system for a few cycles. Such hysteresis will result in intermittent alarms during this period and should be ignored by the health-management scheme. Furthermore, it is important that the health-managers are made aware of any such intermittence that could be expected, before the mitigation actions completely fix the problem.
- Alarms placement: Sensor, Alarm, and Monitor placement are an important issue with software systems as well. While having an alarm for every possible anomaly could help in isolating the fault source, the large number of monitors could easily overwhelm the system. The other extreme of using very few monitors would make it harder to identify and isolate the real fault source. Hence the designers and integrators needing to account for the trade-offs and possibly employ dynamic schemes to deploy and remove monitors as needed.
- Distributed Health Management: Component assemblies that have limited or no interactions and diagnose should be identified and dealt with as independent groups with a different health manager. This will allow the health managers to focus on a smaller region and provide a real-time response to the observed fault effects.

B-3. System Model – Distributed Real-time Managed Embedded Systems (DREMS)

The DREMS platform implementation [B-1] considered in this paper assumes distributed system architecture with mobile computing nodes and an ad hoc mesh network. A specific and singular network access point is not assumed. Several mesh network routing protocols exist [B-16]. This architecture does not constrain the choice of the mesh network. However, we assume that under nominal conditions at least one network route exists between any two nodes in the cluster.

Distributed applications composed from cooperating processes called “actors” provide services for the end-user. Actors are specialized OS processes; they have persistent identity that allows them to be transparently migrated between nodes, and they have strict limits on resources that they can use. Each actor is constructed from one or more reusable components [B-17] where each component is single-threaded, though components can be executed concurrently. Note that though we use the component model described in [B-17] our work is not constrained by this choice and can be applied to other component models as well. The internal architecture of each computing node is described in Figure A. As can be seen from the figure, each layer builds upon the guarantees provided by the previous layer. These guarantees ensures that fault cascades typically travel in only one direction, from a lower layer that provides the service to the higher layer that uses the service. For example, the operating system relies on the guarantees provided by the hardware watchdog which ensures that the system will be reset if a deadlock occurs [B-18].

An operating system kernel provides performance isolation between actors of different applications. This is done by (a) providing separate, protected address spaces per actor; (b) enforcing that a peripheral device can be accessed by only one actor at a time; and (c) facilitating temporal isolation between actors by the scheduler. The temporal isolation is provided via ARINC-653 [B-19] style partitions, which are periodically repeating fixed intervals of the CPU's time exclusively assigned to a group of cooperating actors of the same application. The scheduler guarantees that actors in distinct temporal partitions cannot inadvertently interfere with each other via CPU usage. The encapsulation of peripheral devices into resource manager actors enables fine-grain access control to the resource. Readers are referred to [B-20] for further details on spatial and temporal isolation, both of which are standard mechanisms. Additionally, it provides strict resource monitoring and guarantees that each actor can consume the available computation resource up to the maximum budget that they are allowed.

Actors are divided into two categories: application actors and platform actors. The platform actors provide highly critical services required for the rest of the system to function. Primarily, three kinds of system services are considered:

B-1. Deployment and Configuration Service: It is required to deploy, configure and reconfigure distributed applications. One deployment service instance runs on each computing node.

B-2. Mission Service: The mission service is the operations manager. It is responsible for orchestrating pre-planned timed configuration changes to the distributed architecture. Additionally, this service can re-compute a deployment plan if the currently available computing resources are not enough [B-21].

B-3. Communication (or Cluster) Resource Manager: This service manages the communication resources, i.e. the mesh networks and the routes used. It ensures that the mesh network is connected and that other applications can request an updated mesh status map at any time. It is also responsible for managing the group membership.

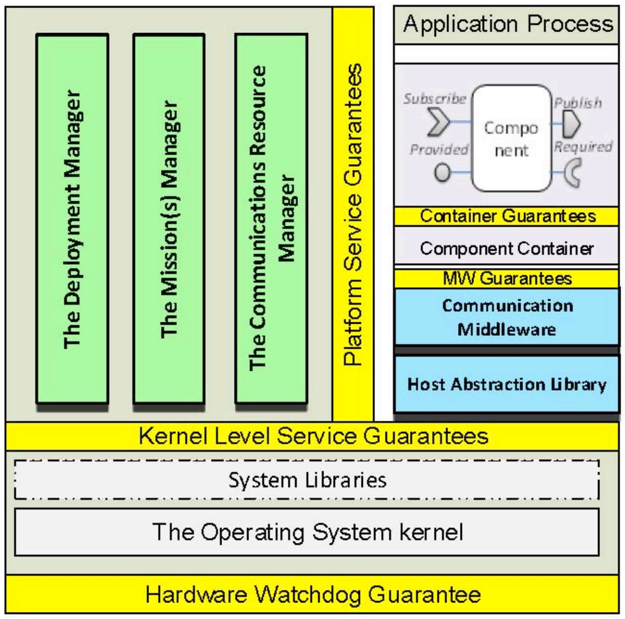


Figure A: This figure shows the internal software architecture of one of the nodes of the distributed system. As shown, each layer builds upon the guarantees provided by the previous layer. These guarantees ensure that fault cascades typically travels in only one direction, from a lower layer that provides the service to the higher layer that uses the service

B-3.1 Distributed Application Model

A distributed software application is a graph of software components that are distributed across processes[2] and hosted on several interconnected computing nodes. Interaction relationships between the components i.e. the edges can be generally grouped into kinds: (a) Peer to peer dependencies and (b) Group interactions.

Peer to peer dependencies, are realized using Facets that are collections of operations (interfaces) provided by a (server) component and Receptacles that are collections of operations required by another (client) component. These two ports can be used to implement synchronous and asynchronous point to point interactions. Generally, a peer to peer dependency implies an order of deployment between components, i.e. the server providing the facet should be deployed before the client with the receptacle.

The group interactions are always asynchronous in nature and do not imply ordered deployment. Group interactions are realized using Publisher and Subscriber ports, which provide a way for components to interact in a global data space defined over Topics. Publisher ports are the gateways to post data samples into the global data space, and subscriber ports notify interested components about arrival of relevant data samples in the space. This feature is provided using the underlying OMG Data Distribution Services (DDS) middleware.

B-3.2 Services

We require the platform to provide the following services.

B-3.2.1 Hardware Watchdog

The hardware watchdog guarantees that it will reset the hardware and thus operating system if a deadlock in the kernel occurs due to which it cannot make any progress. Typically, this situation is result of a critical bug in the kernel code which leads to a state where the task that is responsible for resetting the watchdog periodically cannot be scheduled. This can happen either due to a real-time issue such as priority inversion or can occur due to a critical bug which causes kernel panic and a state of lockup ensues.

B-3.2.2 Operating System kernel

The operating system kernel is responsible for scheduling tasks. It will also ensure that any device failure is communicated to the device managers[3]. The kernel is also responsible for broadcasting messages about change in status of a process to all parties who are interested in receiving that message. Any resource limit violations are also broadcast to managing processes.

[2] Components hosted within a process are located within the same memory space

[3] It is assumed that devices are managed by one process at a time, and this managerial assignment changes infrequently

The kernel also guarantees to restart any critical platform process if it dies.

B-3.2.3 Cluster Resource Manager

The cluster resource manager ensures that a route exists from the node to all other nodes in the cluster. Additionally, it ensures that information about the loss of communication to a computing node is available to any process that is interested and is privileged to know that information. The cluster resource managers are also responsible for maintaining the consensus about group of nodes currently in the cluster. The cluster resource managers guarantee that this information is available from all CRM instances on all nodes. The CRM also provides information about any imminent network outages to any other node in the cluster.

B-3.2.4 State Management Service

The state management service plays an extremely critical role in recovering the functionality of an affected system. While operations such as reset/ restart/ re-deployment (platform or application) would bring the required services alive, the state of the underlying components need to be set correctly for them to restore the desired functionality.

The state management services are responsible for providing the necessary infrastructure to checkpoint and restore the relevant state. The state includes any information that helps restore the lost services such as,

- State information related to the components that are part of the application and/or platform services.
- Static deployment information (per node) that includes the actors (and the components) deployed on each node.
- Dynamic deployment information (per node) that includes the operational state of the components (active/ passive/ inactive), the connectivity of the component ports (connected/ disconnected), and the flow information of the connected ports (the service to which the receptacles are connected).

In order to support state management, the framework should provide the suitable functionality in the middleware that implements component containers. These standard interfaces should support output/ checkpoint and input/ restore the component states based on the relevant component attributes. During check-pointing, the framework code in the container should be able to serialize the attribute data into an octet stream in a Common Data Representation (CDR) format. During restore, the framework provided component container should be able to de-serialize and set the component attributes from the data (octet stream in CDR). The state management service must be configurable in terms of when the check pointing should be triggered for each component or component-assembly -whether it is periodic (at a specific rate) or aperiodic (and based on an event).

While the check pointed data could be stored locally on the node, a better strategy at least for the more critical data might be to distribute this information across the cluster so that the recovery is tolerant to node-specific faults. Further, since the distribution of state-information is a costly operation, the state management service should be configurable in terms of which data is to be distributed and at what frequency. One possible implementation of the state-management service could be done by exploiting the ’Durability’ QoS property in DDS data writer objects that publish the check pointed state information. A PERSISTENT Durability QoS with a history size of 1 would allow the last check pointed data to be available even if the node or the local state management service terminated. A TRANSIENT Durability QoS would also allow the last check pointed state to be available, but it won't be tolerant to faults associated with the local state management service (and hence those of the host-node).

B-3.2.5 Deployment Manager Service

When a distributed application is designed, the developers make assumptions about the deployment of the processes across various computing nodes. These assumptions are represented by virtual node assignments. The task of the deployment and configuration service is to initially set up the system such that all virtual node assignments for an application are mapped to concrete nodes. Nominally, the deployment infrastructure always maps one virtual node to the same physical node. Then, the deployment procedure requires instantiating all components, setting up their port-to-port connections and then activating them. Failures can occur at any time, during the setup or after the application has been activated. A resilient deployment and configuration infrastructure must be able to detect and isolate the failure and then trigger the redeployment and reconfiguration of the affected portions of the application.

The deployment manager service is distributed across the nodes of the cluster. Instances of the deployment managers maintain their own state (currently deployed application) even across resets or shutdowns. The deployment managers must guarantee a transactional behavior which ensures that the state is updated correctly and the application is deployed in a single transaction. Any error should ensure a correct rollback of the changes. Thus, the deployment managers can always be queried to supply the current configuration of the deployed system. Additionally, deployment managers must not block forever waiting for a response from other deployment managers or an application that is being configured.

B-3.2.6 Monitoring Services

Monitoring services play a critical role in facilitating a resilient system. They provide the crucial information about the health of the underlying hardware, the infrastructure, the platform and the application services. This information is useful in detecting the nominal and abnormal states of different aspects of the system. Listed below are some of the monitoring capabilities that would be desirable.

- Distributed Node Health Monitor: The Distributed Node Health Monitoring (DNHM) framework provides information on the health of each node in the cluster. The standard commercial frameworks such as Nagios, Ganglia, Cacti, Munin, Collectd, can be deployed to monitor and collect health-status information from each node and report it to a lead/central node for further analysis and decision making. RFDMon provides a framework that is tolerant to failures in the lead/ central node. The local services on each node can be configured to run scripts that provide information on resource status (CPU, memory, disk, network utilization), variables that indicate the health of hardware (temperature, voltage of Motherboard, CPU), application performance variables (application response time, queue size, and throughput), heartbeats (related to the node and/or specific application processes) and any other conditions specific to the node. It can be configured to report each monitored variable at specific rates with minimum latency and with bounded resource utilization. Most of the monitors described below can be configured to run with the DNHM framework, thereby sharing data and helping to arrive at a consensus.
- Node Heartbeat Monitors: As the name suggests these monitors provide data on the heartbeat

data observed from other nodes in the cluster. This assumes that each healthy node in the cluster is providing heartbeat at a fixed rate.

- Node Resource Monitors: These monitors provide data on the resource (CPU, Memory, Disk, Network, etc.) usage on a node.
- Node Health Monitors: This set of monitors report information that points to the health of the node. It could include data on the temperature and voltage of the motherboard and CPU.
- Device Status Monitors: These include monitors that observe for device failures on the node, e.g. disk failures, network interface card failures, memory failures, etc. These monitors could register with kernel or system libraries and record these errors. Alternately, they could run dedicated self-tests. They could be configured to report specific failures or warn about impending resource limitations, e.g. lack of storage space, memory etc.
- Network Connectivity Monitor: These monitors might collect the information from the cluster resource manager to report the connectivity issues with other nodes in the cluster. Sharing this information with other nodes could help arrive at a consensus.
- Network Quality/Health Monitor: These monitors periodically collect data on properties that qualify the health of a network such as bandwidth available, latency etc. These could be deployed on each node to run on-demand or configured to run periodically on dedicated nodes.
- Platform Services Monitor: These monitors provide important information on the health and operational status of the platform services: deployment manager, cluster manager, state management service, monitoring service, etc. on a node. This could include information on uptime, downtime, anomalies detected, termination/ restart/ reset of any platform service. It could also include details on performance and demand variables specific to the services concerned. These monitors could keep track and report the status/ heartbeat of the monitoring service (and monitors) that are configured to run on a node.
- Application Process Monitor: These monitors observe the health and operational status of the application processes or actors. They are similar to the Platform service monitors described above.
- Timing/Deadline Monitor: These monitors collect and report any deadline timing violations reported by components associated with platform and application services. While the components report such errors to their local health managers, these local health managers might be configured to report to these observations to Timing/ Deadline monitors. These monitors could be part of the Platform Services Monitors and/ or Application Process monitors.
- Application Error Monitor: These monitors collect and report other errors and warning messages detected by application level health managers. The errors collected and reported by these monitors could correspond to persistent observations that are symptomatic of larger system-level problems such as loss of connectivity to a specific node or set of nodes, loss of a specific process etc.
- Restart/ Reset Status Monitor: These monitors report status of reset or restart operation of a node or platform actor or application actor. These could record and report the history,

frequency of resets and restarts initiated, and the result (success or failure) of these operations. These monitors could catch repeated failures to restart a specific process. These monitors work in conjunction with the platform service monitors and/ or application process monitors. These monitors are extremely useful in tracking the status of certain mitigation and recovery actions.

- Event History Recorders: These monitors keep track of the errors and faults detected in specific regions of the system (node, devices, services, components). They record the remedial operations performed (resets, restarts, software updates etc.) and their results. This information could be used in tracking the performance of each element as well as the effect of the remedial actions. They could be particularly useful in analyzing the effectiveness of one or more software updates.

B-3.2.7 Additional Features and Services

Certain other features and services that would be very useful to make the system more resilient are listed below.

- Lead Node Election and Group Consensus: In order to make effective decisions based on the information received from the monitors running on the different nodes in the cluster, it would be useful to have an efficient protocol to determine the lead node of the cluster. This lead node could run additional services that collect data from other nodes to arrive at a consensus and initiate suitable actions to restore lost services. The platform services on the lead node could serve as the lead for each platform service category (deployment manager, fault manger, operations manager, state manager, cluster resource manager, etc.).
- Command Channel: It would be extremely useful if there is a robust fault-tolerant communication channel that is available to provide commands from a central authority (e.g. a ground station) to each node. This could be used by the central authority to issue commands such as to restart the node. Alternately, this channel could also be used by the lead node to communicate any consensus-data (e.g. non-availability of node or network connectivity), to the ground station. This channel could also be used by the ground station to try and restore the network and/or restart a network device.
- Heartbeat service: Each node should provide a heartbeat at a pre-configured rate. If required, platform services and application processes on each node should be configurable to provide a heartbeat.
- Acknowledgement service: Each node should include services that initiate an acknowledgement query and also respond to one. This service could be configurable to query (and respond) on the status of platform services and/or application actor.
- Hardware/ Software Watchdog and Auto Restarts: While the hardware watchdog (mentioned above) is extremely critical to revive a node that seems un-responsive, it is useful only when the problem occurs at the operating system level. Additional software watchdog services may be deployed to automatically restart certain services, or the entire node in certain failure cases. The Linux watchdog command might be useful in these situations. Additionally, certain watchdog services might be run to automatically restart certain critical platform services.
- Auto restart Limits and Broadcast: It would be useful to broadcast a node restart message to other nodes. Also, it could be useful to keep track and limit the number of auto-restarts within a particular time-frame. In case of platform services and application actors, the

Restart/ Reset Status monitor could monitor this limit.

B-3.3 Component Management Interface

Additionally, each component is required to support management interfaces that can do the following:

- Change the operational state of a component. When the component is set to an operational state, it should/ should not perform certain activities. For example, when a component is set to inactive state it should not do any operation. When it is set to an active state it should possibly perform all operations. A passive state could be similar to inactive but the component is likely ready to be switched into active state at any time in future. An additional state could be semi-passive where the component consumes data to update its state, but does not provide any service to other components.
- Connect or disconnect the ports in a component.
- Switch the component from one implementation to another one.
- Accept status information about the health, availability, restoration time of the services that a component's ports are connected to.

B-4. Resilience Scenarios

This section describes various scenarios that affect different parts of the system described in previous paragraph. All scenario descriptions follow a specific format and describe several entries:

- Name: This is a short name that is unique for each scenario. It serves as the title for the section that describes the scenario.
- Description: This provides a narrative description of the scenario.
- Cause: In here, the triggering cause of the scenario is described on some level of abstraction. While the ultimate cause can be a specific physical effect, the 'cause' here means the relevant initiating event that triggers the scenario.
- Location: The location of the initiating cause may include one or more of the following entries:
    - Processing hardware (CPU, memory, etc.)
    - Communication hardware (Network interface)
    - Communication medium (Wired or wireless infrastructure)
    - Operating system kernel
    - C run-time library
    - Middleware -Host abstraction layer
    - Middleware -CORBA: RMI and AMI support
    - Middleware -DDS support
    - Component framework

    - System service -Deployment manager
    - Application component
    - Other

- State: The state of the system when the resilience scenario unfolds is captured here. As the state space of the system can be very large, it is possible that only a description of the state can be given, but not a precise definition.
- Anomaly: This describes the observable anomalies induced by the cause. Anomalies are the observable, detectable, and often measurable consequences of the causes that can be sensed by hardware and/or software sensors. They can be primary (observed in close proximity of the originating cause) or secondary or derived (observed in another component or subsystem, different from the location of the originating cause).
- Effect: Local functional effect induced by the cause are presented here. A functional effect is a change in function, or in the degree the system is able to provide that function. It is not measurable through a physical quantity or by the detection through the presence or absence of an event, but changes one or more functions of the system.
- Impact: Impact refers to the system-level impact induced by the cause. The triggering cause of a resilience scenario can lead to various impacts on the higher level system functions, behaviors, services, etc., if left unmitigated.
- Detection: This describes the location and method of detection of the anomaly/ies. The detection is a decision making process that relies on observations made on the system (either passive measurements or active probing) and draws a conclusion regarding the presence or absence of the relevant anomaly/ies.
- Diagnostics: Location and method of diagnostics of the root-cause of the scenario are mentioned in this entry. The diagnostics is a process that follows detection and determines the cause(s) of the anomaly (ies). The diagnostics results in one or more hypotheses regarding causes that could logically explain the detected anomalies.
- Mitigation: The location and method of mitigation is documented in this entry. The mitigation is a process leading to one or more localized action/s that mitigates the effects of the cause to eliminate or reduce the effects of the cause on the various functions in the system. Mitigation is scoped to one or more subsystems and/or functions.
- Recovery: This corresponds to procedure for system-level recovery. Recovery is a process that leads to the (complete or partial) restoration of system-level functions, services, behaviors, properties etc.

B-4.1 Scenario 1: Computing Node Failure

Scenario 1. Computing Node Failure

Description	The scenario unfolds when a node completely fails and goes silent. The node could be part of a system already under operation or a system under deployment. Some entity in the system is expected to recognize this anomaly, diagnose its cause, determine the actual state of the system and initiate a recovery action. The scenario is a base scenario for other, more specialized scenarios
Cause	A complete and persistent node failure (from the perspective of the rest of the system) could be caused by many different problems, including but not limited to catastrophic failure of the node hardware, failure of the operating system on the node, problems in the network interface card, etc.
Location	The problem could be located in the node hardware or the operating system software deployed on the node.
State	The problem occurs when the system is operation and the system management processes are running on all nodes.
Anomaly	The node is non-responsive to all queries and the node is not sending any data: all communications with the node fail. All nodes in the system observe the same anomaly (i.e. the failure mode is not Byzantine).
Effect	▪ Functions assigned to the node are temporarily or permanently lost. In a distributed application that is partly allocated to the failed node the remaining healthy parts of the application will degrade or fail (depending on the application).
Impact	• There is one less node available for the system to operate, and all resources (including sensors, actuators, devices, specialized devices) directly attached to that failed node are unavailable.
Detection	▪ There are several, non-exclusive options for anomaly detection. ▪ When network communication with the node is attempted, the network stack reports a failure that indicates the loss of connectivity. This detection mechanism works only if the communication is tried. The Cluster resource manager in other nodes would indicate the loss of connectivity with the failed node (Network Connectivity Monitor). ▪ If the nodes monitor each other via a heartbeat mechanism, the lack of the heartbeat message will be detected after a timeout. This detection method needs an active mechanism that keeps testing the connection to the node. If the Node-Health Monitor includes a heartbeat, the other healthy nodes would detect a missing heartbeat from the failed node. ▪ Applications hosted on the failed node become unreachable. The applications on other nodes would report errors through Application Error Monitors on the lack of availability of services hosted on the failed node. ▪ Additionally, failed attempts to get any response to the acknowledgment queries to node, would further indicate a failed or unreachable node.

Diagnostics	▪ Based on the detection mechanisms listed above on each of the healthy nodes, an entity such as an operations manager on each node could conclude that there is a problem with failed node. The healthy nodes could arrive at a consensus to re-affirm that the either there is a problem with the failed node or its network connectivity.
Mitigation	The goal of the mitigation is to re-allocate the functionality assigned to the failed node to other nodes in the system. There are many options possible, including: ▪ Complete relocation: The deployment service may choose to reassign the entire plan that was dedicated to the failed node to another, alternate and healthy node. ▪ Decomposition and reallocation of pieces: The deployment plan meant for the failed node is broken up into smaller plans, each of which is deployed on other healthy nodes. The mitigation process may need state information from the processes that were running on the failed node. This requires the component state be periodically check pointed and shared among the nodes (for an eventual recovery when the failure mode arises). If the failed node has resources that are absolutely necessary then the re-deployment cannot be completed and some higher-level entity must make a change in the system. As part of the mitigation process, all the healthy nodes need to 1 Update their group membership by removing the failed node. 2 Identify the components affected by the failed node. 3 Notify these components about the non-availability of certain interfaces. 4 In case, the failed node was the lead-node of the cluster, the leader-election process should find the new lead-node and updates other nodes. 5 The operations manager in the lead-node should decide on the suitable mitigation action to restore the failed services -complete reallocation or partial reallocation on different nodes. 6 The last check pointed static and dynamic deployment state information of the failed node can be obtained from the state management service. 7 The lead operations manager should provide the updated plan to its local deployment manager. 8 In case the node-failure happened during a deployment process, the lead deployment manager should instruct the healthy nodes to rollback or put-on-hold the incomplete deployment. 9 The lead deployment manager should start the deployment process for

	reallocating the services of the failed node. 10 Once this deployment process is complete, the local deployment managers should be instructed to update affected links.
Recovery	Provided alternate resources are available in the system, the complete system functionality can be restored. System functionality the failed node was contributing to is recovered as the application(s) involved were restarted on other nodes. Note that the state information used in the mitigation may be stale; hence the recovery may result in undesirable transients. If alternate resources are insufficient, some functionality is lost until the failed node is repaired. As part of the recovery process, all the healthy nodes need to 1 Update their group membership with information on any new healthy node added to the network. 2 Reset any new/affected components with the correct check pointed state. 3 Start using any links that have been restored. 4 In case certain services could not be restored, the affected ports should be disconnected. 5 The lead-node should track the health of the restored functionality, through the event-history monitor. 6 The lead deployment manager should start the process to update and complete any of the failed/incomplete deployments.

While the anomalies discussed in this scenario suggest that there are some problems with the node, an assessment of permanent node failure requires records of failed attempts to restart the failed node. The Restart broadcast messages (from the failed node) (observed by Restart/ Reset monitors), and the limits recorded by the Event History Recorders could indicate that the node is not responding despite repeated attempts (up to the set limit). Additionally, a permanent node failure could be confirmed by sending direct commands through the Command Channel to restart the node. If all of the above steps fail to restart the node, then it would confirm a diagnosis of permanent node failure (or permanently unreachable node).

B-4.2 Scenario 2: Node failure during deployment

Scenario 2. Node failure during deployment	
Description	This is a special case for Computing Node Failure described in Scenario 1.
Cause	A complete and persistent node failure (from the perspective of the rest of the system) could be caused by many different problems, including but not limited to

	catastrophic failure of the node hardware, failure of the operating system on the node, problems in the network interface card, etc.
Location	The problem could be located in the node hardware or the operating system software deployed on the node.
State	A deployment process is underway; the deployment procedure has started but has not finished yet. There could be other applications running on the system that could already be partially located on the failed node.
Anomaly	The node is non-responsive to all queries and the node is not sending any data : all communications with the node fail. All nodes in the system observe the same anomaly (i.e. the failure mode is not Byzantine).
Effect	▪ The deployment process cannot be completed due to the failure of one of the required computing nodes.
Impact	▪ There is one less node available for the system to operate, and all resources (including sensors, actuators, devices, specialized devices) directly attached to that failed node are unavailable.
Detection	In addition to the detection described in the base scenario, the Deployment Manager in each of the functional nodes might detect the problem or would be informed about the anomaly.
Diagnostics	▪ Based on the detection mechanisms listed above on each of the healthy nodes, an entity such as an operations manager on each node could conclude that there is a problem with failed node. The healthy nodes could arrive at a consensus to re-affirm that the either there is a problem with the failed node or its network connectivity.
Mitigation	The deployment manager should focus on 1 If necessary, redeploying existing applications (as, presumably, already running applications are more important than newly deployed ones), 2 Continuing the re-deployment of the subject application. The healthy nodes could be allowed to complete their deployment and be made aware of the non-availability or delayed availability of the affected portions of the deployment plan. This would allow a degraded service to start, which would be later restored to normal when the affected portions of the plan are deployed.
Recovery	▪ The interrupted deployment process is completed with one or more alternate nodes being used and the system functionality provided by the new deployment is recovered.

While a deployment activity is taking place, one or more of the nodes involved in the deployment plan may fail, and go silent. The deployment may involve the installation and activation of new software or the reconfiguration of already running applications. The deployment engine recognizes the anomaly, diagnoses its cause and determines the state of the system, and recovers from it. It may rely on other system components, e.g. an operations manager component. Upon, successful recovery, interrupted deployment process is completed with one or more alternate nodes being used and the system functionality provided by the new deployment is recovered.

B-4.3 Scenario 3: Computing Node Failure

<u>Scenario 3. Computing Node Failure</u>	
Description	The scenario unfolds when one or more nodes cannot communicate with the any other nodes in the system. The node could be part of a system already under operation or a system under deployment. Some entity in the system is expected to recognize this anomaly, diagnose its cause, determine the actual state of the system and initiate a recovery action.
Cause	A complete and persistent failure of the network could be caused by many different problems. One problem could be a malfunctioning network hardware that can bring down the network. The network connectivity could be lost due to the communication limits imposed by the physical separation (distance, angle, line of sight) between the communicating nodes in a radio network.
Location	The problem could be located in the Communication medium -the infrastructure associated with the wired or the wireless network.
State	When the problem occurs, certain applications in the system could be operational. It is possible that the software associated with other applications is being deployed, and the deployment process has not yet finished.
Anomaly	▪ The affected nodes cannot communicate with any other node. All requests to/ from the affected nodes cannot be fulfilled. The affected nodes cannot send/ receive data to/from other nodes.
Effect	▪ Functions that depend on communications/ updates from applications hosted on the affected node would be severely affected.
Impact	▪ The affected nodes and any resources (including sensors, actuators, devices, specialized devices) hosted on them are unavailable to the rest of the system.
Detection	Detection In the affected nodes, the anomalies are detected when all the applications encounter problems with the system-calls associated with the network-layer. In other nodes, ▪ When network communication with any of the affected node is attempted, the network stack reports a failure that indicates the loss of connectivity. This detection mechanism works only if the communication is tried. ▪ If the nodes monitor each other via a heartbeat mechanism, the lack of the heartbeat message will be detected after a timeout. This detection method

	coupled with an active mechanism that keeps testing the connection to the node will detect that there is a problem in connecting to these nodes. ▪ Monitors such as the Network Quality/Health monitor and Network connectivity monitor would confirm lack of connection to the affected nodes.
Diagnostics	In case the affected nodes could communicate with the Ground/Base control node through a special network, then each node could relay the observation to the Ground/Base control. Ground/Base control could then diagnose that the nodes are healthy but isolated from the network. If the Ground/Base control has an alternate/ command channel to reach the network hardware, it could ascertain the problem by running additional tests (distance determination etc.). ▪ In case of the healthy nodes, some entity (e.g. an operations manager process) could communicate with its counterparts on other nodes that reach a consensus that the affected nodes are unreachable. This consensus about the group membership is shared among all other healthy nodes.
Mitigation	The mitigation of this scenario requires alternate networks that can be started, or the existence of other dedicated channels (such as TTC) that could be used to communicate with the ground/base control to get a better grasp on the situation Mitigation steps could include one or more of the following steps 1. After diagnosis is confirmed, the affected ports in the deployed components (on healthy connected nodes) should be put in a state that indicates loss of service. This could help the components throw appropriate exceptions. 2. In case a back-up/ alternate network exists, then the ground/base control could bring up the alternate network, which the individual nodes could detect, join and configure. 3. In case the Ground/Base control has an alternate/ command channel to reach the network hardware, it could try to revive the network by restarting the network hardware. 4. If a back-up network device exists, the ground/base control could revive the network, by starting the back-up device. 5. If the problem is associated with distance/ position limitations (direct line of sight), the problem could be resolved by directing the appropriate controls to bring the nodes closer or wait until the nodes are within reach. 6. If alternate services exist, these components should be set to an active state (if required). The affected ports on the healthy nodes should be re-directed/ re-connected to use the services from the alternate components (replicas). 7. If there is sufficient resource availability in the remaining healthy

	connected nodes, then certain critical services (deployed in the affected nodes) can be re-deployed on to the healthy nodes. This would require the lead operational manager to give an updated deployment plan to the lead deployment manager, which could start the deployment with the updated plan
Recovery	1. In case the deployment plan is updated, the states of the newly deployed components should be set by querying the state management service. The affected ports in the healthy component should be re-directed/ re-connected to the newly deployed components. 2. Once service is re-established, either by reviving the network or by re-deploying the essential components on other nodes, it would take some time for the states in the distributed components/ actors/ applications to synchronize. This might require rolling back the states in all components to a previously check pointed state. In any case, the monitoring mechanism should be prepared for a transient period where some problems could exist due to inconsistent states.

B-4.4 Scenario 4: Transient Network Failure

<u>Scenario 4. Transient Network Failure</u>	
Description	This is a special case of network failure discussed in scenario 3. In this scenario, the applications experience a larger than expected communication delay. Transmission of data and completion of requests takes abnormally longer time affecting normal operation. The problem needs to be resolved to recover the system operation.
Cause	An intermittently failing network (communication medium) could be responsible for the high transmission delay. One or more rogue processes could be hogging the network and disrupting the traffic by continuously sending out unexpectedly excessive amounts of data over the network.
Location	The problem could be located in the Communication medium -the infrastructure associated with the wired or the wireless network.
State	When the problem occurs, certain applications in the system could be operational. It is possible that the software associated with other applications is being deployed, and the deployment process has not yet finished.
Anomaly	▪ When the situation unfolds, it would be normal to observe that quite a large number of requests over the network are either being timed-out or are taking significantly longer time to complete. In some cases, the data obtained over the network might not be useful because of the significant delay. If the problems persist long enough, the message-queue could over-

	flow and it is possible that some messages might be lost forever.
Effect	▪ The results output by the distributed applications would be greatly affected. Specifically, it would severely degrade functionalities where timely information sharing among the nodes is critical.
Impact	▪ The capabilities of the system are severely affected because of the degraded network.
Detection	▪ Special monitors (Network Quality/ Health Monitors) that can check the network properties such as bandwidth; latency etc. could detect the problem and make entities such as Operations Manager aware of the situation.
Diagnostics	▪ While the Operations Manager is made aware of the network quality through special monitors, the request-time-outs and aged-message exception seen in the applications could be related to the network quality problem. The Operational Managers could reach a consensus by sharing their observations on network quality.
Mitigation	Additional monitors related to the Node-Health and/or Node-Resource usage could be used to identify if the problem is cause by a rogue node, and/ or if there is any set of processes that are exceeding their network-bandwidth bounds. If the source of the problem is localized to a specific node or to a specific set of processes, these nodes or processes could be isolated or blocked from using the network temporarily. 1. If the problem is perceived to be degradation of the network, the ground/base control could decide to restart the network hardware in the hopes of fixing it. 2. Alternately, if a back-up network exists, it could be restarted to either share some of the load or become the primary network. ▪
Recovery	Once the network is back to normal operational status, the components/ application need to synchronize their states to return to normal operation. Applications might be allowed in a certain order to re-join the network (may be based on priority). This would allow the high priority applications to sync-up before the bandwidth is used by the low-priority applications to sync-up. If the network is revived partially, the operational manager should decide which applications are important and the quota that could be made available to the different application categories so that they can operate in a degraded manner.

B-4.5 Scenario 5: A deadlock in the Kernel

Scenario 5. A deadlock in the Kernel	
Description	We consider the scenario where the operating System on a node freezes as the underlying kernel experiences a deadlock. The built-in hardware watchdog times out and would reset the kernel The entire scenario renders the applications hosted on the node unavailable and affects the system functionality
Cause	The kernel could deadlock due to many reasons -a bug in the kernel, a glitch in one of the hardware devices that the kennel code is not able to handle, a glitch in a device-driver that cascades to freeze up the entire operating system.
Location	The problem could be located in the Operating system kernel and/or in the Processing Hardware -CPU, memory etc.
State	▪ When the problem occurs, certain applications in the system could be operational. It is possible that the software associated with other applications is being deployed, and the deployment process has not yet finished.
Anomaly	▪ When the situation unfolds, the application tasks as well as services hosted on the node freeze. Further, as the watchdog timer trips the node is reset. Any requests to the node are not serviced and any updates from the node are not received by other nodes.
Effect	▪ The services rendered by the applications hosted on the node are completely unavailable if the node deadlocks and the applications are not re-deployed after reset. However, sometimes the services can be intermittently available in a degraded manner if the node keeps resetting.
Impact	▪ Depending on how persistent the problem is, it could affect the functions served by the applications hosted on the node. If the applications come back up seamlessly, it might have limited impact.
Detection	▪ Special monitors/ services (Restart/ Reset monitors) on the node might be configured to send a message to other nodes to make them aware of the reset/ restart.
Diagnostics	▪ The healthy nodes can arrive at a consensus based on the non-responsive node and/or reset message. Further, if the Operational manager on the affected node is restarted automatically after reboot, it can be queried by the other healthy nodes on the restart or it can inform the other nodes about the restart.
Mitigation	▪ The mitigation strategy would involve re-deploying the applications hosted on the node after restart. Also, the other nodes need be made aware of the developments on the affected node (Restart/ Reset monitor). This would include 1. Restart/Reset monitor on the affected node should transmit messages to other nodes about the restart. 2. The Restart/ Reset monitors on the other nodes observe the above message and inform the corresponding operations manager

	3. The operations manager on the healthy-nodes could instruct the deployment manager to inform the affected components so that they do not rely on the services of the affected node. 4. In case alternate/ replica components exist, the deployment manager could re-direct the affected ports to use the services from the replicas. 5. On the affected node,. the operations manager should be restarted automatically. 6. The operations manager might be configured to automatically restart the deployment manager with its previous states on restart. 7. Once all the services are restarted, the operations manager should inform the other nodes about the completed restart. 8. The other healthy nodes might query the operations manager/ deployment manager on the status of the restarts. ▪
Recovery	▪ The recovery process could include The operations manager on the deployment manager about the completed restart on the affected node. 1. The operations manager on the healthy nodes, could inform the deployment manager about the completed restart on the affected node. 2. The deployment managers could instruct the affected component ports to start re-using the old connections (if necessary). 3. The components states might need to be set based on an existing checkpoint. 4. Alternatively, if replicas existed, then the re-started components would need to set their state based on the last check pointed states of the replica components. 5. If required, the replicas can be set back to a passivated state.

B-4.6 Scenario 6: Priority Inversion Leading to a System Reset

<u>Scenario 6. Priority Inversion leading to a System Reset due to watchdog time</u>	
Description	This is a special case of Kernel deadlock. However, this scenario unfolds due to priority inversion in cases where the system is instrumented to monitor the progress of a high priority critical task with a watchdog that can reset the system if the high priority task has not executed for a while. While the reset helps restore the execution of the high priority task, it does not mitigate the source of the problem, but is a temporary solution that addresses the symptoms. If the problem were to manifest often, then this would lead to frequent resets, there by affecting the

	performance of the system. A classic example of this was seen in the Mars Pathfinder where-in a high-priority task was waiting for a shared-resource to be released by a low-priority task. The low-priority task was itself pre-empted by other medium-priority tasks. Whenever this happened, a watchdog timed-out and triggered a system reset/ reboot. In this case, the problem could be solved when the priority inheritance was enabled on the mutex (associated with the shared resource). Enabling the priority inheritance allowed the low-priority task to assume/ inherit the priority of the waiting high-priority task and complete its execution without being pre-empted by the medium priority tasks. Once the mutex was released, the high-priority task could resume its task.
Cause	The real source of the problem in this case is a poor design of the software system that leads to frequent starvation of high-priority critical tasks.
Location	The problem could originate in the critical high-priority tasks associated with the platform/ system service (Deployment/ Operations manager) and/or the application processes.
State	The problem could occur when the system is operational. It could manifest in the platform services when the application software are being deployed.
Anomaly	The common anomaly would be that sometimes certain high-priority tasks do not execute while other tasks (of all priority) seem to be execute as planned. When the high-priority tasks fail to execute, this could lead to problems in other dependent tasks as well as problems in the state evolution/ update. Also, if the system is so configured, one could observe that the system resets itself. The reason for the reset is not quite apparent, unless detailed logs can be generated and manually traced.
Effect	▪ The functions supported by the high priority tasks are possibly affected by this disruption. More so, the system reset can result in other functionalities also being disrupted.
Impact	▪ If priority inversion is the real cause of the problem, the problem is bound to resurface. If the problem is isolated to a specific node, then the node could be put on a cautionary/ watch list.
Detection	If appropriate monitors exist, then they can detect the problem with the scheduling of the high-priority tasks. These could be recorded and/ or reported to other nodes before/ after reset (or other mitigation action).
Diagnostics	▪ It might require a lot of hands-on-debugging -careful walk through of the trace logs to identify the root cause of the priority inversion.
Mitigation	▪ Some of the possible steps to mitigate the problem include 1. Resetting or restarting the affected node when the appropriate watch dog times out. 2. Re-deploying the components and restoring their states to the last check pointed state. ▪ For a more permanent fix,

	1. The lead-operations manager instructs the lead deployment manager to re-deploy the components on to other working nodes. 2. Once re-deployment is complete and/ or replicas exist, the deployment managers instruct the dependent component to re-wire to the replicas or re-deployed components. 3. Apply suitable software updates to the affected node to mitigate the problem. 4. Alternately, if a safe check pointed version exists, then rollback the software to the prior safe version. 5. Re-deploy the components on the updated (or rolled back) node. 6. Inform the operations manager of other nodes, when the task is complete. 7. Instruct the deployment managers to re-wire the affected components back to the original set of components.
Recovery	While the above mitigation steps might help solve the problem temporarily, the Event History recorder should be analyzed to identify if the problems still persist and if a specific re-deployment could mask (or not trigger) the problem.

B-4.7 Scenario 7: Device Failure

Scenario 7. Device Failure	
Description	In this scenario we consider the failure of hardware devices on a computing node. These devices could include the hard drive, memory, network interface card on the affected node.
Cause	▪ The problem is triggered from device failures such as ▪ fault in the hard drive rendering it useless. ▪ lack of storage space in the hard drive. ▪ lack of memory when one or more processes starts hogging/ leaking memory. ▪ fault in the network interface card.
Location	The source of the problem is located in the processing hardware of the associated computing node.
State	▪ When the problem occurs, certain applications in the system could be operational. It is possible that the software associated with other applications is being deployed, and the deployment process has not yet finished.
Anomaly	When the situation unfolds, the appropriate system calls or library calls return with an error code as the OS detects the failure. One or more tasks/ processes on the

	node start failing or operate in a degraded mode
Effect	▪ Certain services (in the node with device failure) as well as their dependent services on other healthy node are affected. The functionalities provided by these services are either unavailable or at-best degraded.
Impact	▪ In the worst case, it is possible that the failure might result in the loss of a node. Otherwise, if it is a temporary glitch then the impact is probably more on the functional level and there is no permanent impact due to the fault.
Detection	The error messages coming from the libraries or the kernel calls provide an indication to the problem. Also special monitors (Device Status monitors) could detect these problems.
Diagnostics	The detection through device status monitors could help in the diagnosis of the specific device faults.
Mitigation	The mitigation strategy would be specific to the resource that is unavailable because of the device failure. 1. In case of memory problems, identifying the terminating processes/ actors that are using excessive memory could help restore memory availability. Alternately, some of the non-essential processes might need to be terminated temporarily to make enough memory available for other critical processes. 2. In cases when there is no storage availability in the hard drive, it might be configured to clear some space. Alternately if additional unused space is available (in the same or new hard drive), it can be configured to be made available to the node. 3. With regards to the network interface card, restarting the device could help restore it. Alternatively, if a back-up device is available, it could be configured to connect to the network. 4. While the above problem is being resolved on the affected node, the operations managers on other nodes should be informed of the problem. The deployment manager on the healthy nodes should inform the affected components and if required disable the affected ports. 5. The lead operations manager could activate replica services that had been put in a passive state. 6. Alternately, if certain services need to be re-deployed on other nodes (either temporarily or permanently), then the lead operations manager should send an updated plan to the lead-operations manager to co-ordinate and complete the re-deployment. 7. The deployment manager in the healthy nodes could be informed of the replica service or the new deployment. 8. The deployment manager could direct the ports on the affected components

	could be redirected/ re-connected to use the service from the replicas. ▪
Recovery	The recovery process could involve operating certain services in a degraded mode, until the affected node is completely recovered. Once the device (and node) is restored, any of the affected components/ processes might be restarted and carefully monitored for their resource consumption.

B-4.8 Scenario 8: Process Failure

Scenario 8. Process Failure	
Description	The scenario deals with process failures, specifically those in which the process terminates unexpectedly. The process could be a platform actor such as a Deployment / Operations Manager or it could be an application actor that is operational
Cause	Any number of reasons could cause a process to terminate. It could be a bug in the code, an unexpected error, an exception that was not caught and handled correctly. Sometimes a process could terminate itself as a design feature that makes it fail silently in certain conditions.
Location	The source of the problem could be any process -an application component, a platform/ system service (Deployment/ Operations manager).
State	▪ The process is typically started and running. It could be any operational state. It is possible that certain application actors can fail during the deployment phase. It is also possible that during a deployment cycle, the deployment manager process terminates unexpectedly.
Anomaly	When a process fails unexpectedly, all dependent processes are affected. They could react to the situation in any of ways including but not limited to ▪ working in a degraded manner ▪ handle/ report the dependency failure in a robust manner ▪ enter a failed / unrecoverable state ▪ time-out ▪ terminate ▪ Moreover, it is possible that if the terminated process was using any shared resource, it could leave the resource (or process) in a hanging/ unusable state.
Effect	▪ The unexpected process termination could lead to non-availability of certain associated services/ functionalities. It affects the dependent processes on same or other nodes. Further it could affect the resources that it was using.

Impact	▪ One or more resources could be temporarily unavailable. One or more services could be temporarily unavailable.
Detection	▪ A special monitor (Application Process Monitor/ Platform Service Monitor) could detect the termination of the concerned processes. Also, Application error monitors could detect errors/ exceptions thrown by dependent components in other processes/ nodes. These application errors could be related to the process termination.
Diagnostics	▪ The detection through special monitors could help in the diagnosis that the specific process terminated and corroborated with the application errors, timing and deadline violations detected on other dependent components.
Mitigation	▪ The Mitigation process involves 1. The Operations manager reading the information from the Application Process Monitor and/or Platform Service Monitor. 2. The Operations manager should validate that this is an unexpected termination. 3. The operations manager instructs the deployment manager to restart the process and redeploy the contained components and set their state to the last check pointed state. 4. Once the terminated service/ process restarts, the restart/ reset monitor broadcasts this information to other nodes. ▪
Recovery	The recovery process involves the following steps 1. The operations manager shares this information with the operations manager of other nodes. 2. The Operations managers in all nodes inform their respective the deployment manager. 3. The deployment manager in the other nodes, use the information to instruct the affected dependent components and disable the affected port or redirect them to an active replica (if exists). 4. Once the operations manager in other undoes are aware of the process restart, the deployment manager re-connects the affected components to the re-deployed components. 5. The state of all the associated components is restored to the last check pointed state.

B-4.9 Scenario 9: Process Failure – Blabbering State

Scenario 9. Process Failure - Blabbering State	
Description	• This scenario describes a process failure wherein the process spits inconsistent data at a rate which is much higher than the nominal expected rate.
Cause	▪ The problem could be caused by a bug in the process code, bug in the code of another process that it depends on, or a bug in a device (sensor/ actuator) that the process is associated with. This could also result when the states of the inter-dependent actors are not correctly synchronized. Other causes include a software update glitch -either in the actor code or with other services that the actor depends on.
Location	The source of the problem could be any process - an application component, a platform/ system service (Deployment/ Operations manager).
State	▪ The process is typically started and running. It could be any operational state. Platform or management actors such as deployment managers/ operational managers could be in the deployment, reconfiguration or operational state.
Anomaly	▪ In this case, the process produces abnormal or inconsistent data/ output. Also, it produces the output at an abnormal rate for a consistently long period of time. The bad data fails the consistency checks/ contracts.
Effect	▪ The functions supported by the actor are affected. The dependent services on other nodes are also affected, compromising some system level function.
Impact	▪ In case some hardware resources are involved, unless the bug is caught early it could damage the devices (actuators) that consume the data/ output. This could affect the resources available.
Detection	▪ Special monitors (Application process monitors, Application Error Monitor, Timing/ Deadline monitor) could observe for problems where the contract guarantees are violated. The Event History Recorder monitor could help identify if the problem has been persistent/ recurring over a prolonged period of time. Additionally, monitors could detect the abnormally data rate of the process if it is significantly different from the design specifications.
Diagnostics	▪ While the special monitors help localize the problem, multiple dependent services can independently detect the violations in contract requirements, and help confirm a problem with the actor. Additional tests on verifying the actor output against one or more independent sources could help confirm them problem.
Mitigation	▪ The mitigation process involves 1. Informing the operations manager about the rogue processes that are the cause for the problem.

	2. Identifying the affected processes based on the Application Error monitors and Application process monitors. 3. The operations manager could instruct the related deployment manager to restart the rogue and affected processes, re-deploy the components and restore their state to the last known checkpoint. 4. Any related hardware resource (sensors/ actuators) could also be restarted.
Recovery	▪ The recovery steps could include 1. Monitoring the mitigation process and allow sometime for the processes to recover and synchronize. 2. Postpone any critical action until the states are synchronized. 3. In case the problems persist, other sources for the problem should be investigated. The rogue processes could be re-deployed on to other nodes to check if the problems persist. Alternately, if a replica service exists, the dependent components could be re-wired to use the replica.

B-4.10 Scenario 10: Bad System State

Scenario 10. Bad System State	
Description	• The scenario captures problems that might occur when a software update is applied to the platform code and/ or application code. It is possible that while the software update solves issues that were present with the earlier version, it introduces new unintended problems/ bugs in other features/ aspects. It is also possible that the software update cannot work smoothly with other tools already installed in the node. This could lead to problems such as deadline violations, inconsistent results, unavailable services etc.
Cause	The problem occurs when the updated software does not work correctly with other parts of the existing system.
Location	The problem originates in the layer where the software update is applied. It is quite possible that the problem does not manifest in the place where the software patch is applied. It could manifest in a related component with which the updated software interacts or it could manifest in a completely unrelated entity.
State	▪ The problem could occur when the system is operational. It could manifest in the platform services when the application software is being deployed.
Anomaly	While there isn't any specific anomaly to watch for, it is quite possible that depending on the nature of the fault introduced different category of anomalies could be observed. These could include contract violations, deadline violations, process failures, or sometimes even node failures.
Effect	▪ Depending on the extent of the problem and the usage of the updated

	component/ service, the effect could be minimal and localized to a specific functionality or affect multiple functionalities and nodes.
Impact	▪ It is quite possible that the software update could bring down specific desired functionalities or render the update nodes as unusable.
Detection	The anomalies could be detected by the monitors built for the specific violations in the platform services and / or application services. These could be part of the Platform Service monitors or Application process monitors that identify that there are problems with certain platform services and / or application actors. The errors observed in the processes could be monitored by the Application error monitors, Deadline /Timing Violation monitors.
Diagnostics	▪ While the diagnosis in this case could point to unrelated faults (due to the nature of the anomalies), it might be possible to rule out these problems through additional tests. A look at the past record of the problems prior to the update could reveal that a bug in the updated software could be the culprit. A record of the Event History recorder would help identify when the problems were present and when they were not. If it was due to a software patch, it could be easily observed that the problems were seen after certain patch was applied.
Mitigation	▪ Some obvious mitigation strategies could include 1. The operations manager could instruct the deployment manager to start alternate/replica processes where the components are dependent on known safe versions of the software. 2. The dependent components could be re-wired to use the components based on older software version. 3. The processes using the faulty version are terminated. 4. If any fix is available in future, the components could be re-deployed with the updated software library.
Recovery	The recovery process as in other cases, involves restoring the component states to a prior check pointed state. Additionally, the component states need to synchronize. The observations from the Application Process Monitors, the Platform Service monitor, the Application Error monitors, the timing/deadline monitor etc. should be recorded by the Event History Recorder and tracked for any violations from the rolled back or updated version. ▪

B-4.11 Scenario 11: Security Violation

Scenario 11. Security Violation	
Description	▪ The scenario captures problems that might occur when the software system is compromised because of a security breach.
Cause	The problem occurs when there is a security breach
Location	▪ The security breach could affect any application or platform service.
State	▪ The problem could occur when the system is operational. It could manifest in the platform services when the application software is being deployed.
Anomaly	▪ While there isn't any specific anomaly to watch for, it is quite possible that depending on the nature of the attack different category of anomalies could be observed. These could include denial of service, loss of communication bandwidth, incorrect system operation, loss of equipment etc. While some of these anomalies could be observed, it is quite possible that the security breach could go undetected.
Effect	▪ Depending on the extent of the attack, the effect could be minimal and localized to a specific functionality or affect multiple functionalities and nodes.
Impact	▪ It is quite possible that the security violation could bring down specific desired functionalities or render the one or more nodes as unusable, thereby bringing down the entire mission.
Detection	▪ The detection of the security breaches can be performed by using specific trained detectors based on a threat model. Such detectors are trained to understand the nominal system and detect any activity that is outside of the nominal state. It is quite possible to train such systems for detecting intrusion attacks, denial of service attacks etc. Further, it is possible to design security policies and create observers that check for any violations to the security policies. These violation-observers could serve as an initial monitor to signal a possible security violation. Also, based on the system design to thwart such attacks, queries or messages could carry appropriate labels that are suitably encrypted. Any non-conformity to the correct label set could indicate a possible violation.
Diagnostics	▪ The diagnostics for the security violation is obtained by using the output of the special detectors and observers that are designed for the purpose as well as any abnormalities that are witnessed in the system. This could help understand which part of the system is being attacked by the security breach.
Mitigation	In some cases, when the attack is isolated to a specific service, it would be appropriate to shut-down the service for the time being and inform any affected services. Alternately the service could be migrated and restarted on another node.

	If the attack is localized to a specific set of nodes(s), it should be possible to isolate the appropriate node(s) from the network and re-configure any affected functionality on alternate nodes. For services that have a active or passive replica on another node, the traffic could be redirected to the replica. The affected parts of the system should be made aware of the mitigation action and the downtime to restore the service.
Recovery	The recovery in this case would imply identify the possible loopholes/ bugs in the system and fixing them with appropriate software and/or policy update.

B-5. Conclusion

This document outlines a possible architecture for resilient systems and the possible resilient system level services (deployment manager, operations manager, communications/ cluster resource manager) that would be key to making the system resilient. It discusses the various monitoring services that could be deployed in the system to observe for specific anomalies and use them isolate and thereby diagnose the appropriate fault candidate. It discusses specific services to support recovery mechanisms (re-configuration through deployment manager, group-consensus and leader election, state management service). It discusses various scenarios where the system functionality is affected and the resilience strategies that could be developed based on the detection, diagnosis, and mitigation and recovery services deployed / available in the framework.

Acknowledgement: This report has been prepared for the Resilient Software Systems project that has been supported by the Air Force Research Laboratory under Award FA8750-13-2-0050.

B-References

[B-1] T. Levendovszky, A. Dubey, W. Otte, D. Balasubramanian, A. Coglio, S. Nyako, W. Emfinger, P. Kumar, A. Gokhale, and G. Karsai, “DREMS: A model-driven distributed secure information architecture platform for managed embedded systems,” 2013.

[B-2] Avizienis, “Fault-tolerance: The survival attribute of digital systems,” Proceeding of the IEEE, vol. 66, no. 10, pp. 1109–1125, Oct. 1978.

[B-3] J.-C. Laprie, C. B´eounes, and K. Kanoun, “Definition and analysis of hardware-and software-fault tolerant architectures,” Computer, vol. 23, no. 7, pp. 39–51, 1990.

[B-4] M. R. Lyu, Software Fault Tolerance. John Wiley & Sons, Inc, 1995, vol. New York, NY, USA.

[B-5] Avizienis and J. Kelly, “Fault tolerance by design diversity: Concepts and experiments,” Computer, vol. 17, no. 8, pp. 67–80, Aug. 1984.

[B-6] Randell and J. Xiu, “The evolution of recovery block concept,” Software Fault Tolerance, pp. 1–21, 1995.

[B-7] Randell, P. Lee, and P. C. Treleaven, “Reliability issues in computing system design,” ACM Comput. Surv., vol. 10, no. 2, pp. 123–165, 1978.

[B-8] Avizienis, “The n -version approach to fault tolerant software,” IEEE Transactions on Software Engineering, vol. 11, pp. 1491–1501, December 1985.

[B-9] J.-C. Laprie, “Dependable computing and fault tolerance: Concepts and terminology,” in Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing,’ Highlights from Twenty-Five Years’, June 27–30 1995, p. 2.

[B-10] S. S. Brilliant, J. C. Knight, and N. G. Leveson, “Analysis of faults in an n-version software experiment,” IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 238–247, 1990.

[B-11] Brown and D. Patterson, “Embracing failure: A case for recovery-oriented computing (ROC),” High Performance Transaction Processing Symposium, 2001.

[B-12] D. A. Patterson, “Recovery oriented computing: A new research agenda for a new century,” HPCA, vol. 00, p. 0247, 2002.

[B-13]] A. B. Brown and D. A. Patterson, “Rewind, repair, replay: three r’s to dependability,” in EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC. New York, NY, USA: ACM Press, 2002, pp. 70–77.

[B-14] ——, “Undo for operators: building an undoable e-mail store,” in ATEC’03: Proceedings of the USENIX Annual Technical Conference 2003 on USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1.

[B-15] R. Hanmer, Patterns for fault tolerant software. John Wiley & Sons, 2013.

[B-16] I. F. Akyildiz and X. Wang, “A survey on wireless mesh networks,” Communications Magazine, IEEE, vol. 43, no. 9, pp. S23–S30, 2005.

[B-17] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13), Paderborn, Germany, Jun. 2013.

[B-18] D. J. Lu, "Watchdog processors and structural integrity checking," Computers, IEEE Transactions on, vol. 100, no. 7, pp. 681–685, 1982.

[B-19] Document No. 653: Avionics Application Software Standard Inteface (Draft 15), ARINC Incorporated, Annapolis, Maryland, USA, Jan. 1997.

[B-20] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose, "A Software Platform for Fractionated Spacecraft," in Proceedings of the IEEE Aerospace Conference, 2012. Big Sky, MT, USA: IEEE, Mar. 2012, pp. 1–20.

[B-21] S. Pradhan, W. Otte, A. Dubey, A. Gokhale, and G. Karsai, "Towards a Resilient Deployment and Configuration Infrastructure for Fractionated Spacecraft," in Proceedings of the 5th Workshop on Adaptive and Re- configurable Embedded Systems (APRES '13), CPSWeek. Philadelphia, PA, USA: IEEE, Apr. 2013.

APPENDIX C: TRADE-OFFS IN FAILURE MANAGEMENT STRATEGIES

Towards understanding cost trade-offs in failure management strategies for space missions

Nagabhushan Mahadevan, Abhishek Dubey, and Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, 37212.

C-1. Introduction

Engineering space systems is challenging for straightforward reasons. In addition to the remote nature of the systems, unforeseen phenomena can affect the system at any time. Given the cost and effort required to launch and establish these systems to orbit, it is understandable the resilience is one of the most important requirement. One of the cornerstones of resilient design for any system is ensuring fault-tolerance. A popular and well-established strategy adopted for improving the resilience in large space system architectures is to deploy redundant components with design diversity to tackle faults in critical subsystems [C-1], [C-2]. This strategy is coupled with extensive shielding or hardening techniques to protect the on-board electronics from radiation exposure in outer space. There are extensive guidelines on establishing the hardness assurance of space electronics depending on the scope and length of the mission [C-8], [C-10].

Space Missions, in general, can be classified depending on the criticality of their missions. On one end of this spectrum are missions that can tolerate no downtime or extremely minimal downtime. Depending on the nature of the mission, some of them can tolerate longer downtimes provided the system eventually becomes available. The traditional techniques of redundancy with comparison or acceptance based testing are extensively used in highly critical missions with requirements for extremely high availability. The redundancy based techniques mask certain class of persistent and transient faults that may develop in one or more, but not in all of the redundant components at the same time, thus ensuring that faults do not lead to eventual system or subsystem failures. These systems also require massive investment to design and develop techniques that improve the reliability of individual hardware components, for example by radiation hardening. While these techniques based on redundancy and hardening are often applied in space system design, the costs involved are prohibitive, which then limits the possibility of launching multiple independent systems[4]. Rather the design is restricted to monolithic systems with several redundant subsystems.

Systems on the other end of the spectrum are not very critical and can tolerate reasonable downtime provided the functionality is eventually restored. In these systems the cost is lowered by ensuring that the systems are not individually engineered with high availability. Alternative fault management strategies are available for such systems which depend on anomaly detection, fault diagnosis, and reconfiguration to restore system functionality. These strategies often

[4]Probability of failure of two independent systems is the product of probability of failure of each of the individual system, thereby increasing the chances of mission success

employ a model-based comparison approach at runtime to detect anomalies, often referred to as online model-checking. The model-based approach also extends towards diagnosis, and reconfiguration. They can support detection, diagnosis and recovery from a broad class of faults and untoward events and can be dynamically updated to support changes in the mission/ system. However, the performance and fault coverage of these online model-checking and reconfiguration techniques is dependent on the correctness of the model and the support infrastructure (sensors, monitors, diagnosis algorithms, schedulers, reconfiguration and deployment services, etc.). Additional resources are required to operate the model-checking and reconfiguration strategies. Given the dynamic nature of these methods, understandably, such online management is not used widely in space systems.

With the advent of recent developments to adopt many small and cheap fractionated satellites in place of the traditional single monolithic costly satellite, there is a need for architectures to support critical tasks with minimal or no downtime and employ alternate fault-management strategies that do not depend exclusively on dedicated redundant resources. They should be able to handle faults identified at design time as well as unforeseen events discovered at runtime, allow reconfiguration to restore or add functionality. With this in mind, this report looks at cost trade-offs to be considered in designing hardware and software failure management strategies in large system architectures for space mission applications.

The outline of the report is as follows. The next section focuses on the variants of redundancy based fault management techniques. Section B-3 deals with the radiation exposure in space, its impact on electronic components and the hardness assurance process employed to deal with this problem. Section 0 describes the working of the fault-management strategies that depend on fault detection, diagnosis, mitigation and recovery through online model checking and reconfiguration. It discusses the software infrastructural support required to implement these alternative fault-management strategies. Section C-5 discusses the costs, benefits and risks associated with choosing each of these strategies when designing resilient system architecture.

C-2. Redundancy based Fault Tolerance Strategies

The redundancy based fault tolerance aims to improve the reliability of the system by using redundant parts, with the assumption that failure of a part is an independent event. Hence, the failure probability of the overall subsystem is lower as it is a product of the failure-probability of the individual parts. The redundancy based techniques use a comparison (i.e. a ‘Voter’) or acceptance check (i.e. an ‘Acceptance Test’) based scheme to decide if the part is working correctly and pass on the ‘correct’ output to the downstream subsystem.

A voter based redundancy scheme, can tolerate up to n-2 independent faults in the subsystem with n redundant components. The redundant parts, whose output is compared by a voter, could be active replicas with variations introduced in terms of design, implementation, or temporal execution.

In a diversified design, several variants of the same software are used with an acceptance test employed at the end to compare the output from each of the variant. The rationale behind this approach is the expectation that components built differently will fail differently [C-3]. A diversified design has at least two variants plus a decider, which monitors the outputs of the variants. Three such strategies for software fault-tolerance are: the Recovery Block approach [C-4], [C-5], N-version programming approach [C-6], and the N self-checking programming approach [C-1], [C-7].

The recovery block technique [C-5] uses the checkpoint and restart technique to recover from a fault. It uses multiple versions of the same software module as alternates. Upon failure of an acceptance test, the checkpoint state is used to restart the computation by using the next alternate version. In this approach, at least two versions of software module are required. If all alternates fail, the module issues an exception to the rest of the system.

In the N-version programming approach, multiple versions of same programs is executed in parallel, with a voter selecting the output most likely to be correct. This approach is different from the recovery-block approach in that it does not require an application dependent voter and that it needs at least three versions of the same module to work. However, the parallel execution necessitates the need to ensure input consistency. In 1990, Brilliant, Knight and Leveson published results from a large-scale experiment conducted in N-version programming [C-8]. In the experiments, they prepared twenty seven versions of a program at two different universities and then executed them one million times. The results of the experiment were intriguing. They noticed that different versions were found to be reliable when used individually with only a small number of failures. However, when they correlated these failures across different versions they discovered that the number of input cases when more than one version failed was significantly large. Upon post failure analysis, they further discovered that correlated failures arise from logically unrelated faults in different parts of the algorithms. They hypothesized that most of the faults resulted from the fundamental flaws in the algorithms that the programmers designed. Therefore, changing the development tools or methods to create new versions did not reduce the number of correlated failures in N-version software.

N-Self-checking approach [C-1], [C-7] is a combination of Recovery block approach and N-version approach. This approach has two variations. The first variation uses acceptance test for each version as in the recovery block approach, however, the acceptance test for each version can be different. By executing these versions in parallel, this approach enables switching of output in case of errors instead of restarting from previous checkpoint. The other variant uses a comparison technique. It groups the variants into set of two, with a comparison unit forwarding the result to a selection unit only if the two versions in a set produce identical results. The selection logic then selects the outputs from different version similar to the N-version technique. The drawback of using this technique is the possibility of running into situation where both versions in a set produce identical wrong output.

The dedicated active redundancy schemes based on simultaneous parallel executions guarantee extremely high-availability. In case of redundancy through sequential repetitive computation, a worst case execution time is factored in to the design. Both techniques guarantee availability even in the presence of broad class of independent persistent and transient faults in the redundancy components. Of course, this comes at a cost of additional dedicated resources for redundant operation, as well higher cost towards other aspects such as subsystem design, launch (payload size, weight), power usage, thermal issues etc. Also, this strategy could limit the scope for on-demand adaptation and reconfiguration to meet some unforeseen operational challenges and/or updates to mission requirements. Moreover, while this strategy is extremely effect when there are up to a certain number of independent transient and persistent faults in the redundancy components/ execution schemes, it would require additional support to handle common failure modes that affect all the redundant units.

C-3. Radiation and Platform Shielding

The space radiation environment can lead to extremely harsh operating conditions for on-board electronics and the related systems. The radiation environment could change completely based on the date, time and duration of the mission being considered. The primary radiation sources of concern include energetic particles trapped by earth's magnetic field forming the Van Allen radiation belts in the magnetosphere, the cosmic rays transiting the solar system and the solar particle events produced by solar flares [C-8]. The radiation belts include electrons and protons with energies over 30 keV, while the cosmic rays include energetic heavy ions extending to energies beyond TeV. The solar flares produce primarily energetic protons with energies ranging up to hundreds of MeV. The interaction of the above environments with satellite structural and shielding materials also results in the generation of secondary radiation.

The effects of these primary and secondary radiations on space missions are varied. They include degradation of electronic performance due to accumulation of total ionizing doze as well as single event phenomena. Single event phenomena broadly include three different effects in electronic components. Single Event Upset (SEU) produces bit flips in electronic components. These are transient and do not cause any permanent damage. Single-event latch up (SEL) include one or more bit flips occurring together (and may be being persistent), resulting in the part ceasing to function, and drawing excessive current. Operation is restored when the device is turned off and then back on. The excessive current drawn can destroy the device if the power supply cannot handle the current. Finally, in case of single-event burnout (SEB), the device fails permanently.

A rigorous methodology is employed to ensure that the parts and components used in the space system meet the required Radiation Design Margins (RDM) and thereby are not compromised by the radiation environment. This is often referred to as Hardness Assurance Process [C-10]. This involves understanding the system requirements and environmental definitions to identify the RDM required for each part and thereby the system. This is followed by part selection and testing to characterize the parts in terms of expected RDM. This could be followed by design, testing and development efforts to improve radiation hardness through shielding and radiation tolerant design. Shielding could involve spot shielding of the specific part or the shielding of the entire box.

The radiation hardening significantly improves the reliability and tolerance to radiation effects as compared to COTS parts [C-8]. It thereby greatly improves the reliability and fault-tolerance of the entire system. The process does come at a steep cost in terms of design, manufacturing and launch costs. In this context it is worth noting that the launch cost to geostationary orbit is \$50,000 /kg, so even a few grams of shielding incur significant cost. Additionally, changes to the design to achieve radiation tolerance could impose additional penalties in terms of greater power consumption and poorer performance as compared to the COTS components. Since these parts are produced one-off and not for the mass market, the scope for price reduction based on the market demands is non-existent. Processes to create incremental updates to the designed components to keep pace with the COTS components are not economically viable. Further, the significant investment in building these radiation hardened components also implies that significant operational time to recover the cost. This leads to use of older technology that does not keep pace with the performance of COTS technologies, sometimes resulting in resource limitations.

C-4. Online Model checking, Fault-repair and Reconfiguration

Online model checking refers to the process of checking the operational status and detecting any anomalous behavior by comparing operational data against those expected in some (possibly active) model of the system. The model could be as simple as a constant threshold stored in a look-up table, or a discrete behavioral model, or mathematical models of varying degrees of complexity. The process of detecting anomalies is followed by diagnosis to identify the fault cause(s) responsible for the anomalies observed. Finally, mitigation and recovery to restore functionality is achieved through fault-repair and reconfiguration strategies. This approach can be configured to account for anomalous behavior and their cascading effects due to faults identified at design time as well as latent bugs, common mode failures or other unforeseen events or attacks that disrupt the nominal operation. Also, this can be applied to augment the system reliability when redundancy based fault tolerance strategies are already in place.

The anomaly monitors could be based on observing different aspects of the system such as heartbeats of the computing nodes and applications, watchdogs associated with the hardware and software operation, resource utilization of the applications hosted, application data etc. These are occasionally compared against preset values or thresholds, model outputs or expected behaviors.

The diagnosis schemes could use the status of these anomaly monitors to localize and isolate the fault source(s) based on a table look-up, or using rule-based or model-based reasoning. They could employ a hierarchical approach as well as consensus-based schemes between multiple independent observers.

Fault repair and recovery strategies could be based on simple reactive schemes where a specific set of pre-determined actions are utilized to mitigate the fault effects and restore the functionality. Alternately, based on the available resources more deliberative, search-based strategies based on constraints could be employed to identify the best solution to repair and restore the system.

While the approach could be configured to handle disturbances (pre-determined and unforeseen) from a wide-array of potential causes (faults, glitches to updates, security violations), its success depends on the robust and reliable operation of multiple actors; i.e. anomaly monitors, diagnosis and fault localization engines, and recovery and reconfiguration services. This involves allocation of dedicated resources towards these tasks, as well as the overheads associated with any support services that might be required to monitor resources used (and available), dynamic deployment of tasks based on the available resources, state managers for re-starting state-based services (through checkpoint and restore). The systems need to designed, tested, verified and validated according to the specifications and requirements and overheads in terms of resource and system and subsystem downtime needs to be well understood.

C-4.1 Recovery-Oriented Computing

In [C-11], Brown and Patterson state that the heterogeneity and complexity involved in most large-scale service systems inherently leads to unforeseen failures. Therefore, in Recovery-Oriented Computing (ROC)[5] [C-11], [C-12], [C-13] they concentrate on reducing time to recover from faults and thus offer higher availability and aims to reduce total cost of ownership. ROC emphasizes on testing recovery systems and helps make recovery procedures a holistic part of the architecture rather than a patch or add-on that leads to extra complexity. They suggest multiple techniques [C-13] in support ROC, including redundancy to remove single points of failure, system partitioning for fault containment, aid and testing of diagnosis and recovery mechanisms.

C-4.2 Model Based Software Health Management

As part of a research effort on Model-Based Software Health Management (sponsored by NASA's Aviation safety program) we adapted a diagnosis scheme used for Systems Health Management and applied it towards diagnosis of a software component assembly. Foundation of the architecture is a real-time component framework (built upon an ARINC-653 platform) that defines a specific model of computation for software components [C-14]. This framework brings the concept of temporal isolation, spatial isolation, and strict deadlines from ARINC-653 and merges these with the well-defined interaction patterns described by the CORBA Component Model [C-15].

The real-time health management scheme involved a two-level software health management scheme: a Component-level Health Manager (CLHM) provided localized and limited service for managing the health of individual software components, and a System-Level Health Manager (SLHM) managed the health of the overall system. The SLHM used a diagnosis engine to reason about fault effect cascades in the system, and isolates the fault source components. This was possible because the data and behavioral dependencies and hence the fault propagation across the assembly of software components could be deduced from the well-defined and restricted set of interaction patterns supported by the framework. Once the fault source is isolated, the SLHM applied either a pre-determined reactive scheme or a deliberative search based scheme to reconfigure and recover the system [C-21].

While the work was effective in showcasing the applicability of the online-model checking, repair and re-configuration strategy, it also shed light on the issues that need to be considered

[5] http://roc.cs.berkeley.edu/

while deploying such strategies [C-20]. This included among other things accounting for local and system-level effects of mitigation and reconfiguration actions, issues related to monitoring anomalies (timing, intermittents, false positives and false negatives), fault-masking effects and the fault-containment regions enforced by fault-tolerance schemes designed into the system.

C-4.3 Resilient Distributed Real-time Managed Systems

More recently, efforts to design and develop a resilient software infrastructure for deploying and managing distributed real-time embedded systems led to a better understanding of the underlying functional, design and runtime requirements to support an online mitigation and recovery strategy [C-19]. Some of the requirements include

- A distributed deployment service that is resilient to faults in the deployed application, support services, computing nodes and network
- An operational management service that employs additional services to be aware of operational status, resource usage and availability status, health status of all the hardware and software services, and status of recovery operations.
- A group consensus and leader election scheme to support the health-management and recovery efforts across the cluster.
- A distributed state preservation (checkpointing) and restoration service to aid in the recovery of applications that are state dependent.
- Multiple monitoring services at the level of hardware, operating system, middleware and data distribution framework, applications and their associated components.

C-5. Analyzing trade-offs for designing fault-tolerant architectures

Like in any other case, the trade-offs in applying different fault protection schemes in the system architectures should be assessed from the perspectives of the costs involved, the benefits accrued and the continued risks associated with applying the specific set of strategies.

In the past, a number of approaches have been proposed along to assess the cost-benefit analysis of model based diagnostic system [C-16], integrated vehicle and systems health management (IVHM) [C-17], [C-18]. These works look at the benefits in terms of the revenue generated from the availability of the system, captured in terms of the product of availability and revenue generated per unit time of availability. They penalize the design in terms of the cost of detection and cost of risk. The detection cost relates to the cost of development of the IVHM and the related maintenance and life-cycle costs. The risk cost is related to any failures that might be introduced and the consequent downtime with its share of loss in revenue.

Additionally, while designing systems to support IVHM, [C-17] considers fitness functions to evaluate the trade-offs involved in deploying different kinds of monitors and implementing related fault-detection and diagnosis algorithms. The fitness function is evaluated for each fault and then summed over all the possible faults. For each fault, the downtime cost, the cost associated with downstream failures, the cost of the health management technology, the associated risk reduction by using the health management technology, and the risks introduced due to incorrect diagnosis (including false-positives) are taken into account.

Analyzing the value proposition associated with the different fault management strategies in the system architecture, all the pros and cons need to be looked at from the perspective of

Costs: This will include the cost of designing, development, testing, validation and certification (if required) of the relevant technology in the context of the problem that it is trying to solve. Additionally, it will include the operational cost in terms of the additional resources required to deliver the fault-tolerance/ recovery functionality.

Benefits: The benefit would be estimated against the possible set of faults that can be managed, the criticality of these faults towards hindering the success of the mission, the availability guaranteed. It would also take into account the benefits of the mitigation action in containing the fault and protecting the system.

Risks: The risk should take into account failures in the fault management technology based on the confidence in its underlying concepts and framework and the related consequence in terms of system or functionality downtime.

Mission Specification and Requirements: The assessment of the cost-benefit trade-offs is highly dependent on the use-case or the specific mission requirements. In the case of space systems, this would involve consideration of the length of the mission (long or short term), the requirements of re-use across mission, the need to handle flexible requirements be adaptable to changes in requirements. The mission specification can factor in if single monolithic satellites are to be used or multiple relatively cheaper fractionated satellites are to be deployed. These issues would help decide the contexts in which alternate fault tolerant schemes can be explored.

Design Assumptions, Functionalities and Faults: In order to better understand the cost-benefit analysis of the different fault-tolerance strategies for a specific mission, it is important to understand

1. The assumptions related to (1) system design, (2) fault monitoring design, (3) fault mitigation design, (4) redundancy design, (5) hardening design.
2. The possible set of faults in different categories should be considered. These include known and/or expected faults based on Failure Mode Cause and Effect Analysis (FMECA) on the hardware components, the radiation induced effects (single event, total ionizing dose), software issues related to hardware faults and misoperations, latent software bugs, software update issues, network errors, security problems, and system issues such as transient scheduling problems. For each fault, information of its downstream effect, its severity, associated cost and fault-containment area from testing, experience and expert knowledge would be helpful.
3. The required functionalities (mission specific applications, support services), their criticality, the services they are dependent on, and their downtime tolerance.

Comparison in different phases of the life-cycle: The tasks involved in different phases of the mission life cycle could be factored into the cost-benefit trade-off analysis.
The design and development phase has to deal with

- Systems engineering costs related to design, verification, validation and certification of the fault management strategy.
- Hardness assurance costs to estimate the required radiation design margin and identify the process to ensure that the parts satisfy the requirement
- Manufacturing costs of radiation-hardened hardware

The launch, operational and maintenance phases have to deal with

- Costs associated with the weight of the shielding, power usage, additional hardware resources
- Costs to run additional operations that support the fault management strategies including redundant computation, comparison or acceptance based testing, monitoring, online model-checking, diagnosis, fault-repair and reconfiguration operations.
- Cost to update the software and maintain the hardware.

An initial assessment of some aspects of the costs, benefits and risks of the fault management technologies in accordance with the factors mentioned above can be based on the promises, limitations, published literature and past experience associated with each of these technologies.

Redundancy based fault tolerance schemes are known to work with minimal downtime and when used with n-modular redundancy scheme they can tolerate up faults in up to n-2 redundant parts.
Also, the use of redundancy brings down the failure probability (multiplicative effect) and thereby increases the reliability of the system. The techniques do not focus on specific set of faults, but are applicable to a broad class of independent, persistent (or transient) faults. The approach does not help when the issue is related to a common failure mode across all redundant components. This breaks the assumption of independent faults and it is quite possible that the problem does not show up in comparison or acceptance tests. From a risk assessment standpoint, it would be important to understand the sources of these common failure modes and their occurrence (failure) rate. The design process should be able to ensure that this failure rate of common mode faults is extremely small. The extra cost in terms of dedicated redundant resource usage, weight, power should be factored in to assess the number of redundant parts that can deliver the most benefits.

Radiation hardening through careful parts selection, very strict manufacturing guidelines, shielding and redesign of the electronic circuity ensures that the subsystems meet and exceed the radiation design margin requirements. This increases the reliability of the components to operate without any downtime. While the hardness assurance process is prohibitively costly, additional attention in keeping the shielding to a minimum would help reduce payload size and launch cost. The performance penalty should be kept in mind while redesigning to harden the circuitry.

Commercial technologies exist where radiation hardening is kept to a bare minimum while using redundant COTS components. In one such case, COTS processors execute on the same instruction set in a clock-synchronized environment and the output is compared by a Voter circuit. Radiation hardening is applied to the most critical element in the chain, i.e. a Voter circuit. Another strategy employed is to modify the processor to execute each instruction at least twice and validate consistent output with a rad-hard Voter component. In case of a fault, the third execution is performed to choose the output. Active Testing and software scrubbing are other techniques employed to identify and reverse bit flips resulting from SEU and SEL.

Online model checking, fault repair and reconfiguration claims to be more flexible in terms of the disturbance set (fault, update glitches, security violations) that it can handle and is not restricted to a prescribed set of faults. Further, since resources are not tied down with dedicated redundant computation, the architecture could support changes in mission requirements. However, additional resources are required to support monitoring, diagnosis, reconfiguration and there is a non-zero downtime before the service is restored.

A rigorous process needs to be applied to understand the limitations and assumptions made in the model-based approach. With the dynamic nature of these systems, it calls for careful design and implementation to limit the false-negative and false-positive detections. The costs, benefits and risk associated with the operation of each component, subsystem and service should be assessed taking into account the total set of individual faults as well as the number of simultaneous faults that can be tolerated. Also, the effectiveness of the recovery or redundancy mechanism should be studied in the context of the function criticality, and its downtime tolerance, as well as the additional resource requirements.

In extremely mission critical and dedicated long term requirements without much adaptation, a radiation hardened redundancy based approach would allow for lower downtime across the longer life span. In case of non-critical mission requirements where there is support from other independent fractionated satellites that can be easily deployed on-demand, a slightly relaxed approach towards radiation design margins and radiation hardening can be applied. COTS hardware that meets the relaxed RDMs can be used on the fractionated satellites.

The online checking and reconfiguration strategy could work well in cases where the functional requirement is not mission critical, can tolerate well-defined downtimes and there is need for adaptation including on-demand services. However, it needs to be applied on small well-defined systems where its operation can be well understood, verified and validated.

C-6. Conclusion

The report outlined the traditional fault management strategies based on redundancy and radiation hardening as well as alternate strategies based online model-checking to detect anomalies, diagnose faults, repair and reconfigure to restore functionality. It discusses the pros and cons of the strategies, the specific mission requirements, the cost factors and the risks that need to be considered while evaluating the trade-offs in applying each scheme to enable resilience to space systems. Since the techniques can coexist the best course solution would be to find an ideal mix of the different strategies based on the mission specification, requirements and budgetary limits.

Acknowledgements: This report has been prepared for the Resilient Software Systems project that has been supported by the Air Force Research Laboratory under Award FA8750-13-2-0050.

C-References

[C-1] J.-C. Laprie, C. B´eounes, and K. Kanoun, "Definition and analysis of hardware-and software-fault-tolerant architectures," Computer, vol. 23, no. 7, pp. 39–51, 1990.

[C-2] M. R. Lyu, Software Fault Tolerance. John Wiley & Sons, Inc, 1995, vol. New York, NY, USA.

[C-3] Avizienis and J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," Computer, vol. 17, no. 8, pp. 67–80, Aug. 1984.

[C-4] Randell and J. Xiu, "The evolution of recovery block concept," Software Fault Tolerance, pp. 1–21, 1995.

[C-5] Randell, P. Lee, and P. C. Treleaven, "Reliability issues in computing system design," ACM Comput. Surv., vol. 10, no. 2, pp. 123–165, 1978.

[C-6] Avizienis, "The n -version approach to fault tolerant software," IEEE Transactions on Software Engineering, vol. 11, pp. 1491–1501, December 1985.

[C-7] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in Proc. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 'Highlights from Twenty-Five Years', June 27–30 1995, p. 2.

[C-8] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "Analysis of faults in an n-version software experiment," IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 238–247, 1990.

[C-9] J. Wertz and W. Larson, Space Mission Analysis and Design, ser. Space Technology Library. Springer Netherlands, 1999. [Online]. Available: http://books.google.com/books?id=veyGEAKFbiYC

[C-10] J. Barth, K. LaBel, and C. Poivey, "Radiation assurance for the space environment," in Integrated Circuit Design and Technology, 2004. ICICDT '04. International Conference on, 2004, pp. 323–333.

[C-11] Brown and D. Patterson, "Embracing failure: A case for recovery-oriented computing (roc)," High Performance Transaction Processing Symposium, 2001.

[C-12] A. Patterson, "Recovery oriented computing: A new research agenda for a new century," hpca, vol. 00, p. 0247, 2002.

[C-13] Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (roc): Motivation, definition, techniques,," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep., 2002.

[C-14] Dubey, G. Karsai, and N. Mahadevan, "A component model for hard real-time systems: CCM with ARINC-653," Software: Practice and Experience, vol. 41, no. 12, pp. 1517–1550, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1083

[C-15] N. Wang, D. C. Schmidt, and C. O'Ryan, "Overview of the CORBA component model," in Component-based software engineering: putting the pieces together. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 557–571.

[C-16] J. Kurien and M. D. R-Moreno, "Costs and benefits of model-based diagnosis," in Aerospace Conference, 2008 IEEE. IEEE, 2008, pp. 1–14.

[C-17] J. Kacprzynski, M. J. Roemer, and A. J. Hess, “Health management system design: Development, simulation and cost/benefit optimization,” in Proceedings of the IEEE Aerospace Conference. IEEE, 2002.

[C-18] Y. T. C. Hoyle, A. Mehr and W. Chen, “Cost benefit quantification of ISHM in aerospace systems,” in ASME International Design Engineering Technical Conference, 2007.

[C-19] Karsai, D. Balasubramanian, A. Dubey, and W. Otte, Distributed and managed: Research Challenges and opportunities of the next generation cyber-physical systems, in 17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014, 2014, pp. 1{8. [Online]. Available: http://dx.doi.org/10.1109/ISORC.2014.36

[C-20] N. Mahadevan, A. Dubey, and G. Karsai, Architecting Health Management into Software Component Assemblies: Lessons Learned from the ARINC-653 Component Model," in ISORC, 2012, pp. 79-86.

[C-21] N. Mahadevan, A. Dubey, D. Balasubramanian, and G. Karsai, “Deliberative, search-based mitigation strategies for model-based software health management," Innovations in Systems and Software Engineering”, vol. 9, no. 4, pp. 293-318, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11334-013-0215-x

LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

AADL	Avionics Architecture Description Language
ACE	Adaptive Computing Environment
AFRL	Air Force Research Laboratory
CIAO	Component-Integrated ACE ORB
CLP	Constraint Logic Programming
DANCE	Deployment and Configuration Engine
DB	Database
DM	Deployment Manager
DoD	Department of Defense
DREMS	Distributed Real-time Embedded Managed Systems
GEO	Geosynchronous Orbit
GME	Generic Modeling Environment
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ISIS	Institute for Software Integrated Systems
LEO	Low-Earth Orbit
MDE	Model-driven Engineering
MIC	Model-integrated Computing
NC	Network Calculus
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
OSD	Office of the Secretary of Defense
RAE	Resilience and Analysis Engine
RDM	Resilient Data Model
ReSoSML	Resilient Software Systems Modeling Language
SMT	Satisfiability Modulo Theory
TAO	The ACE ORB
TMR	Triple-Modular Redundancy
UML	Unified Modeling Language
XML	Extensible Markup Language